RealTime Feedback in Coding Games

Major Qualifying Project

Written By:

John Amaral
Shuxing Li
Grace Pelella

Advisors:

Lane Harrison
Charlie Roberts

A Major Qualifying Project
Worcester Polytechnic Institute

Submitted to the Faculty of the Worcester Polytechnic Institute in partial fulfillment of the requirements for the Degree of Bachelor of Science in Computer Science.

August 22, 2019 - May 13, 2020

ABSTRACT

Visual feedback in the space of interactive media can give users more insight into the execution of a program. This paper discusses the implications and results of incorporating visual annotations into both programming environments and computer games and how it affects the user experience. We combined both of these spaces and developed a coding-based resource-management game and used playtesting feedback to analyze the impact of the visuals on a user's performance in the game. The collected data shows that in general, the introduction of visuals assists players in making more well-informed decisions within a time limit. Based on these gathered results, there is clearly a lot of undiscovered potential for visual annotations in both coding and gaming to provide more informative experiences.

Table of Contents

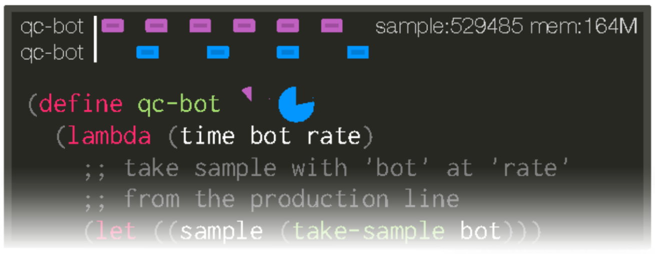
List of Figures

Executive Summary

User experiences are shaped by the intentional design of a program and how well it communicates its current state to the user. This applies to almost any form of media, and it is especially characteristic of programs such as coding environments and computer games. For a programmer, it can be very challenging to understand the execution of written code without any sort of immediate feedback. As a gamer, having the ability to respond to in-game events and complete multiple different tasks within a relatively short time frame is often a requirement to progress in the game. In either case, some intermediary between the program and the user is needed to provide more context and allow the user to interact with the program on a deeper level. One simple and effective way to maintain communication and flow between man and machine is with the use of visual feedback.

The intention of these visuals is to aid the user in better understanding the software they are working with. They can be found in all different shapes, sizes, and colors, which usually depends on their functionality. Some can be complex graphs that reveal patterns and metrics within the program's data, while others are as simple as small dots or color coding alert the user to when changes are occurring. No matter what form they take, all of these visuals give the user greater insight into the inner workings of the program they are utilizing.

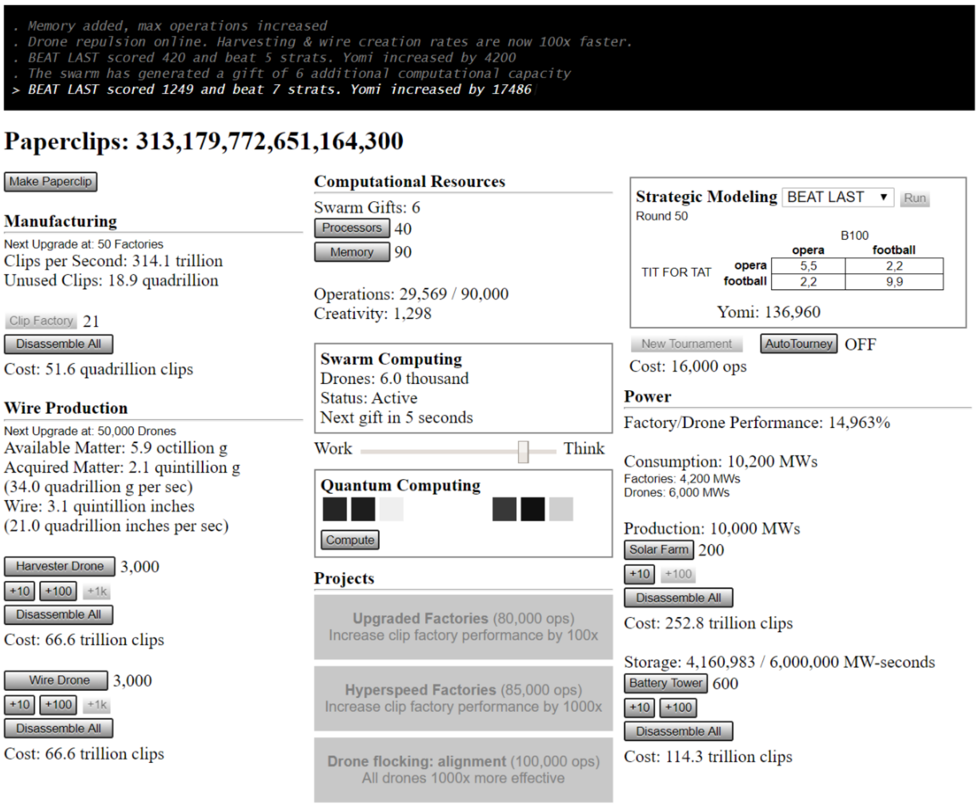
During our background research, we discovered many examples of these types of visuals. We found that in the programming space specifically, visual annotations strike a balance between the state of the world and the state of the code. In other words, these visuals clearly communicate how each line of code is executed and how program data is manipulated while also keeping the programmer informed of the program's effects on real-world resources such as time and memory. **Figure 0.1** below exemplifies just one scenario of coding visuals from the Impromptu programming environment [1]. In this specific example, the circular visual in-line with the function definition ("define qc-bot") represents the time left in executing that specific line of code.



```
qc-bot                                          sample:529485 mem:164M
qc-bot

(define qc-bot
  (lambda (time bot rate)
    ;; take sample with 'bot' at 'rate'
    ;; from the production line
    (let ((sample (take-sample bot)))
```

Figure 0.1: *Screenshot from the Impromptu IDE, where simplistic visuals such as the timer clearly indicate the state of the code to the programmer [1].*

We also found several examples of visual feedback being used to provide a better gameplay experience for players. Video games typically put a lot of effort in designing their interface to ensure players have enough information to succeed in playing the game, and oftentimes this is done with simple visual indicators representing changes in the data within the game. This is especially the case with resource-management games, where several different statistics need to be managed at once. One great example, although the visuals are not too complicated, is in **Figure 0.2** below with the browser game Universal Paperclips [2]. The game introduces assets to manage one at a time in their own separate areas of the computer screen to not overwhelm the player. Although the visual feedback is not as apparent as the previous example, each resource is clearly distinguishable from the others due to carefully-crafted layouts and visual elements that make each component distinct.



```
. Memory added, max operations increased
. Drone repulsion online. Harvesting & wire creation rates are now 100x faster.
. BEAT LAST scored 420 and beat 5 strats. Yomi increased by 4200
. The swarm has generated a gift of 6 additional computational capacity
> BEAT LAST scored 1249 and beat 7 strats. Yomi increased by 17486
```

Paperclips: 313,179,772,651,164,300

Manufacturing

Next Upgrade at: 50 Factories
Clips per Second: 314.1 trillion
Unused Clips: 18.9 quadrillion

Clip Factory 21
Cost: 51.6 quadrillion clips

Wire Production

Next Upgrade at: 50,000 Drones
Available Matter: 5.9 octillion g
Acquired Matter: 2.1 quintillion g
(34.0 quadrillion g per sec)
Wire: 3.1 quintillion inches
(21.0 quadrillion inches per sec)

Harvester Drone 3,000
Cost: 66.6 trillion clips

Wire Drone 3,000
Cost: 66.6 trillion clips

Computational Resources

Swarm Gifts: 6
Processors 40
Memory 90

Operations: 29,569 / 90,000
Creativity: 1,298

Swarm Computing
Drones: 6.0 thousand
Status: Active
Next gift in 5 seconds

Quantum Computing

Projects

Upgraded Factories (80,000 ops)
Increase clip factory performance by 100x

Hyperspeed Factories (85,000 ops)
Increase clip factory performance by 1000x

Drone flocking: alignment (100,000 ops)
All drones 1000x more effective

Strategic Modeling BEAT LAST
Round 50

B100

		opera	football
TIT FOR TAT	opera	5,5	2,2
	football	2,2	9,9

Yomi: 136,960

AutoTourney OFF
Cost: 16,000 ops

Power

Factory/Drone Performance: 14,963%

Consumption: 10,200 MWs
Factories: 4,200 MWs
Drones: 6,000 MWs

Production: 10,000 MWs
Solar Farm 200
Cost: 252.8 trillion clips

Storage: 4,160,983 / 6,000,000 MW-seconds
Battery Tower 600
Cost: 114.3 trillion clips

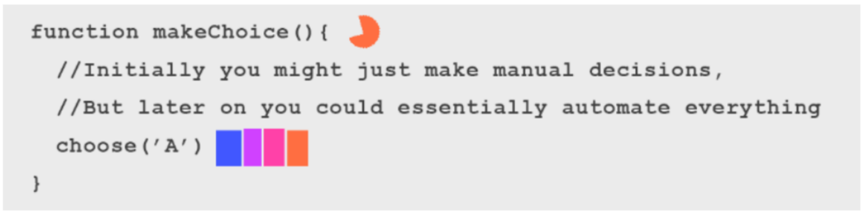
Figure 0.2: *Screenshot from the Universal Paperclips game, where resource management is made easier with simple visual elements and separation of the layout design [2].*

For this MQP, our goal was to develop a programming-based computer game that took advantage of the capabilities of visual feedback by supplying the player with enough knowledge to make decisions and manage time efficiently. Our general process flow followed these four main steps:

1. Brainstorming game mechanics that took advantage of inline visual annotations
2. Designing several screen layouts to maximize visual communication
3. Developing the game in code and implementing visual elements
4. Testing gameplay and the effects of visuals on players for data analysis

We devoted a lot of time early on to thinking about different potential concepts for our coding-based game and how visual feedback could be tied to gameplay. We eventually came up with the idea of having several statistics (hunger, food, security, population, military, and science) to keep track of, and prompts within the game would force players to choose between two options that each affected the statistics differently. The prompt structure allowed us to envelop the player in a narrative about traveling through time and needing to balance resources using visuals elements presented within a computer terminal.

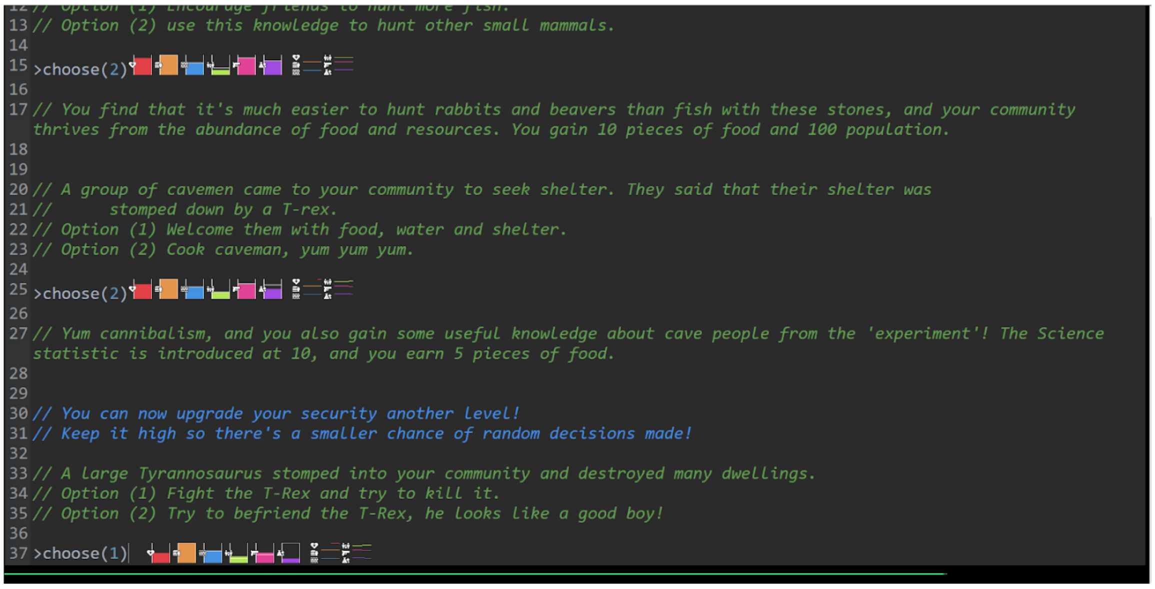
We spent a considerable amount of time designing several possible screen layouts and iteratively improving them to match our game mechanics and goals. As we got closer to finalizing our design for the game, all of the elements became more compact until all of the player action took place within one window in an online browser. This constrained our visual design to small annotations in-line with the code that users typed, which better aligned with the previous research we conducted and also with our project guidelines. **Figure 0.3** below shows a design mockup with some examples of inline visuals we considered. The bar graphs representing each statistic value did make it into the final game, whereas the timer icon next to the first line was changed to be a bar scrolling from right to left.



```
function makeChoice(){
  //Initially you might just make manual decisions,
  //But later on you could essentially automate everything
  choose('A')
}
```

Figure 0.3: *Portion of one of our final design mockups with some examples of code that players could write and visual elements they could utilize.*

When we were ready, we began official development of the game. The game was coded in JavaScript and utilized a library called CodeMirror [15] that lets users enter lines of code in a webpage as if they were using an actual programming environment. We implemented different commands, such as choose() to decide on an option in response to a prompt, eat() to deplete some food but increase your hunger, and secure() to increase your security when prompted to do so. The most complicated but experimental mechanic we introduced was the automate() command, which let users write their own functions that updated with each frame of the game. This allowed them to automate processes that would otherwise require their attention to execute manually. With all of these additions, the resource and time-management coding-based text adventure *Coding Through the Ages* was born! **Figure 0.4** below shows a screenshot from the final version of the game.



```
// Option (1) Encourage friends to hunt more fish.
// Option (2) use this knowledge to hunt other small mammals.

>choose(2)

// You find that it's much easier to hunt rabbits and beavers than fish with these stones, and your community
thrives from the abundance of food and resources. You gain 10 pieces of food and 100 population.


// A group of cavemen came to your community to seek shelter. They said that their shelter was
//      stomped down by a T-rex.
// Option (1) Welcome them with food, water and shelter.
// Option (2) Cook caveman, yum yum yum.

>choose(2)

// Yum cannibalism, and you also gain some useful knowledge about cave people from the 'experiment'! The Science
statistic is introduced at 10, and you earn 5 pieces of food.


// You can now upgrade your security another level!
// Keep it high so there's a smaller chance of random decisions made!

// A large Tyrannosaurus stomped into your community and destroyed many dwellings.
// Option (1) Fight the T-Rex and try to kill it.
// Option (2) Try to befriend the T-Rex, he looks like a good boy!

>choose(1)
```

Figure 0.4: *Screenshot from the final version of Coding Through the Ages.*

In the back-end of the game, we recorded gameplay data such as statistic values and time at each decision for any run-through of the game and stored it using Firebase. This was set up for the purpose of playtesting, where we set up Zoom sessions with 18 different players and had them try *Coding Through the Ages*. We kept track of all of their gameplay data, wrote down their immediate reactions to game elements using think-aloud note-taking techniques, and collected their opinions in a brief survey at the end of the game. As compensation for testing, each one of them received a $10 Amazon eGift card. More details on playtesting can be found in the survey questions, study protocols, and informed consent form in **Appendices A, B,** and **C** respectively.

From all the data we collected, we were able to perform data analysis and draw general conclusions about the game based on graphs we made (some using the D3 JavaScript library, which can be found here, and others from the Google forms survey) and the survey responses we gathered. We found that the vast majority of players had trouble in the first few stages of the game and generally tended to lose during their first run. However, players overall got better and better with each playthrough, and most of them managed to finish by the end of their playtesting sessions. We did notice as players went through the game that some of the prompts appeared to be poorly balanced based on results of certain decisions made by the players, especially regarding changing the population or science statistics.

Based on observation of the players and their statistics over time, we believe that the bar chart, the timer visual, and the highlighted text were extremely effective in communicating the changes in game data to the player. However, other visuals such as the sparklines and value change indicator were rarely used by the players and did not seem effective in helping the players make decisions. Some players made decisions based on their interpretation of the prompt while others attempted to speed their way through the game by choosing seemingly-random options, and players who took more time to read prompts tended to have more trouble completing the game in the amount of time allotted than those who self-evaluated as a fast reader.

Using valuable feedback from the players and some features we had to cut due to time constraints for the project, we drafted out some future directions to take the game in. We would like to expand upon the automation functions to make them a more user-friendly tool that the players can experiment with, such as by allowing the players to parse prompts and make decisions automatically based on the parsing results, or giving players the ability to create their own variables. Another potential idea would be to make the coding experience in the game more like an actual command line with options such as hitting the up arrow instead of re-entering previously-entered commands and having a cheat table as a sort of command API to remind players about specific mechanics without using the man() command. Finally, one aspect we wanted to include but did not have enough time for was having a branching narrative, where different decisions players made would lead them to completely different prompts than other players. All of these additions to *Coding Through the Ages* would make it an even more unique coding and gameplay experience.

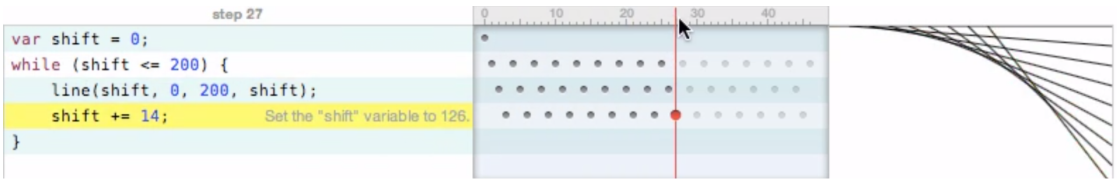
Overall, we are extremely satisfied with how *Coding Through the Ages* turned out. According to our testers, it managed to be a fun coding-based game with visuals that clearly showed the current state of the game and led to better informed decisions being made. Although plenty of additional features could be introduced to improve the experience, we feel that we adequately met our project goals, and we hope that maybe someday another MQP group could continue work on the game or take it in a new direction. Click here to try the game out for yourself. Click here for the MQP presentation, and click here to watch the gameplay trailer.

Chapter 1

Introduction

Interpreting the current state of a computer program has always been a difficult challenge for users to overcome. Whether that program is a coding environment where a programmer is typing their own lines of code, a video game that a player is trying to complete, or any other form of digital interactive media, understanding each component and how it changes over time is crucial to creating a valuable user experience. When limited information about the execution of the program is presented to the user, or when the information does not provide enough context into the inner workings of the program, there is a large disconnect between the user and the program that results in overall less usability and more frustration than should be needed. There are many ways to help resolve this problem, and one of the most effective ways is with visual elements that are used to represent the program's data and how the data is modified as a result of user actions and time.

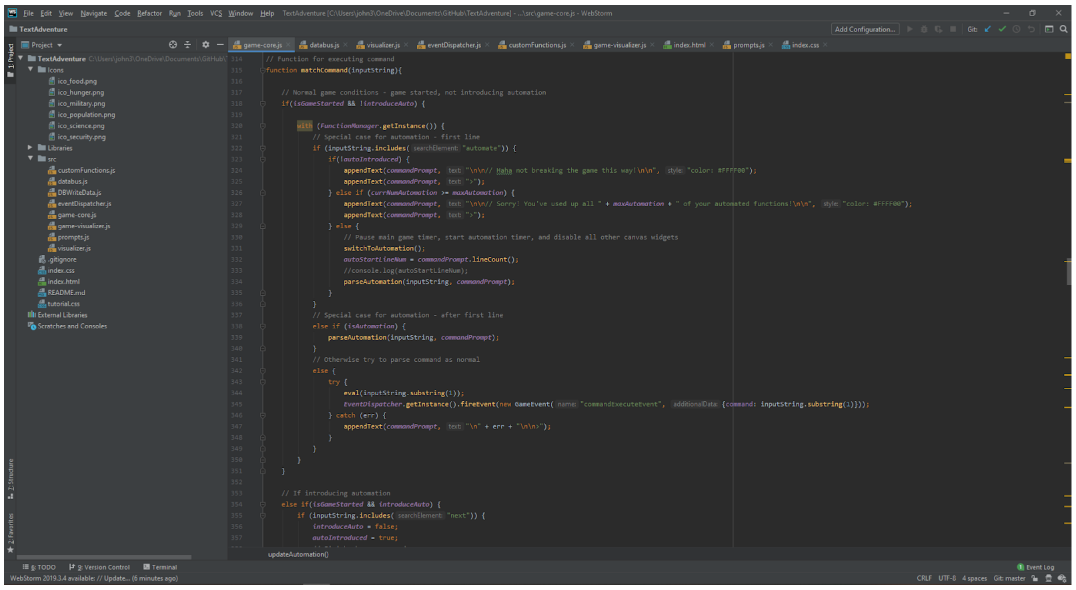
One applicable use case for these visuals is within the context of coding. Supplementary visuals, also referred to as annotations, in programming environments can make it easier for programmers to better understand how the code within a program will run. These annotations visually represent the data being manipulated within the code, or provide a direct view into the current state of the running program. The visuals can have multiple forms, such as informative graphs, color coding, simple icons, and comments generated in real-time, depending on the type of input or output the program takes or produces. For instance, in Brett Victor's [3] learning example in **Figure 1.1** below, each inline dot to the right of the code represents when a certain line of code is executed within the while() loop, and the flow of the code can be observed by reading left to right in the diagram. To the right of those visuals, the actual output of the program (lines being drawn at different angles, or "shifts" in this case), is also visualized depending on what dot (point of execution) the user has selected.



```
var shift = 0;
while (shift <= 200) {
    line(shift, 0, 200, shift);
    shift += 14;
}
```

Figure 1.1: *An example of visuals that can show both the data and state of execution within a program.Click here for the full video [3].*

Figure 1.1 is a great example of providing more context for the user to understand the flow of the code, and therefore most programming environments must have similar visuals, right? Unfortunately this is not necessarily the case: many commonly used integrated development environments (IDEs) are lacking these concise and informative visuals. However, many modern IDEs have basic visual elements, such as color coding to distinguish between comments, function declarations, and variable references, and syntax highlighting for showing warnings or errors in certain locations within the program. Take a look at **Figure 1.2** below, which is a screenshot of the Jetbrains Webstorm IDE for JavaScript development: is it possible to understand what the program does within just a few seconds of viewing the code, even with the color coding?

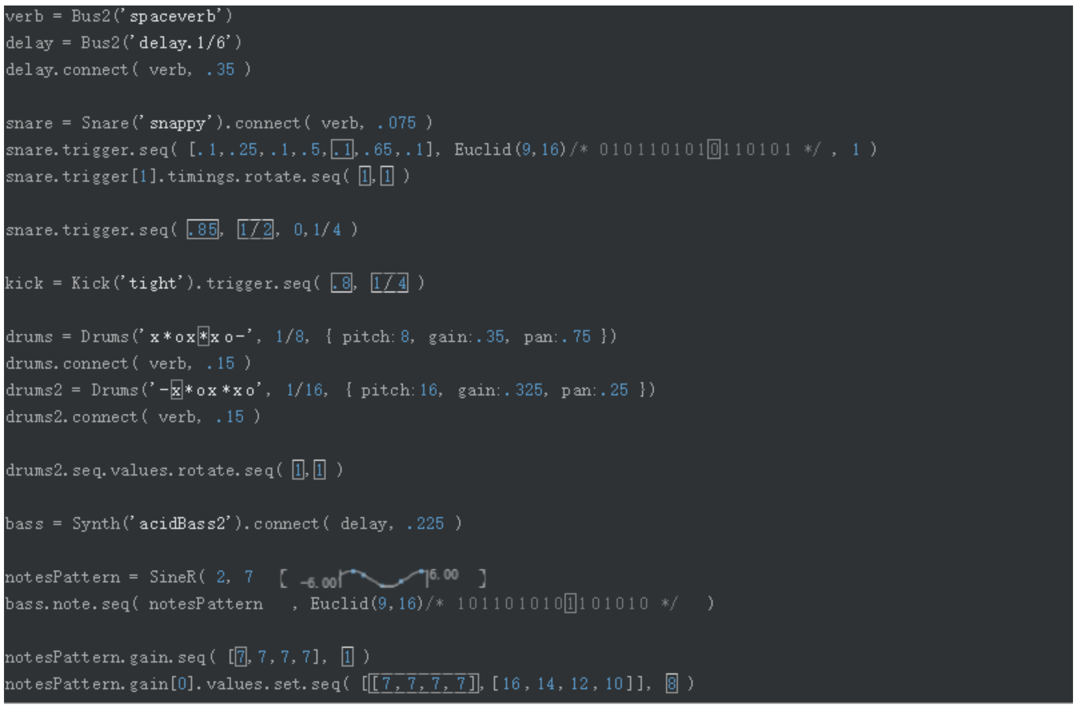


```
// Function for executing command
function matchCommand(inputString){

    // Normal game conditions - game started, not introducing automation
    if(isGameStarted && !introduceAuto) {

        with (FunctionManager.getInstance()) {
            // Special case for automation - first line
            if (inputString.includes( "automate")) {
                if(!autoIntroduced) {
                    appendText(commandPrompt, "\n\n// Haha not breaking the game this way!\n\n", "color: #FFFF00");
                    appendText(commandPrompt, ">");
                } else if (currNumAutomation >= maxAutomation) {
                    appendText(commandPrompt, "\n\n// Sorry! You've used up all " + maxAutomation + " of your automated functions!\n\n", "color: #FFFF00");
                    appendText(commandPrompt, ">");
                } else {
                    // Pause main game timer, start automation timer, and disable all other canvas widgets
                    switchToAutomation();
                    autoStartLineNum = commandPrompt.lineCount();
                    //console.log(autoStartLineNum);
                    parseAutomation(inputString, commandPrompt);
                }
            }
            // Special case for automation - after first line
            else if (isAutomation) {
                parseAutomation(inputString, commandPrompt);
            }
            // Otherwise try to parse command as normal
            else {
                try {
                    eval(inputString.substring(1));
                    EventDispatcher.getInstance().fireEvent(new GameEvent( "commandExecuteEvent", {command: inputString.substring(1)}));
                } catch (err) {
                    appendText(commandPrompt, "\n" + err + "\n\n>");
                }
            }
        }
    }

    // If introducing automation
    else if(isGameStarted && introduceAuto) {
        if (inputString.includes( "next")) {
            introduceAuto = false;
            autoIntroduced = true;
```

Figure 1.2: *A screenshot of code we wrote within the Webstorm IDE, with some color coding [4].*

For most people, especially those who are not used to code structure and syntax, it would be very difficult to understand what the code in **Figure 1.2** does just by observing it quickly for a few seconds or even skimming through it. Although the color coding distinguishes between the individual elements of the code, it does NOT communicate the type of data used or how the program processes and iterates upon that data. If the focus of the visuals was on communicating the functionality of the code rather than the organization of it, it would be much easier to understand how each line adds to the workflow of the program.

Many individuals and organizations have gone out of their way to incorporate helpful visual annotations into their programs, and some of these programs even require the visuals to be present at all times. By making the visual annotations a necessity, these programs focus on providing a more comfortable user experience. Take Charlie Roberts' live coding program Gibber [4] for example. In a live coding program, the user can enter code and have the output change in real time, which is what Gibber does to play different musical notes and instruments depending on the code typed in. The visuals within Gibber are essential to the user experience: the notes within the code are highlighted when they are currently being played, and sine waves representing the patterns of notes are displayed in line. **Figure 1.3** below shows a screenshot of Gibber currently playing a song.



```
verb = Bus2('spaceverb')
delay = Bus2('delay.1/6')
delay.connect( verb, .35 )

snare = Snare('snappy').connect( verb, .075 )
snare.trigger.seq( [.1,.25,.1,.5,.1,.65,.1], Euclid(9,16)/* 0101101010110101 */ , 1 )
snare.trigger[1].timings.rotate.seq( 1,1 )

snare.trigger.seq( .85, 1/2, 0,1/4 )

kick = Kick('tight').trigger.seq( .8, 1/4 )

drums = Drums('x*ox*xo-', 1/8, { pitch:8, gain:.35, pan:.75 })
drums.connect( verb, .15 )
drums2 = Drums('-x*ox*xo', 1/16, { pitch:16, gain:.325, pan:.25 })
drums2.connect( verb, .15 )

drums2.seq.values.rotate.seq( 1,1 )

bass = Synth('acidBass2').connect( delay, .225 )

notesPattern = SineR( 2, 7  [ -6.00        6.00  ]
bass.note.seq( notesPattern   , Euclid(9,16)/* 1011010101101010 */   )

notesPattern.gain.seq( [7,7,7,7], 1 )
notesPattern.gain[0].values.set.seq( [[7,7,7,7],[16,14,12,10]], 8 )
```

Figure 1.3: *A screenshot of Gibber playing a song, with visuals to indicate the change in musical data [4].*

The constant presence of visuals in Gibber allows the user to successfully manage and recognize multiple tasks at once. Based on this concept of juggling multiple tasks, it is clear to see how these types of visuals could be applied to digital games. As previously mentioned, having concise visuals in computer games leads to a more enjoyable player experience, and it is almost a necessity in games that, like Gibber, require the player's attention to be divided between multiple components at once. A great example of this application done right is in the resource-management game Reigns [5], where the player needs to make decisions as the ruler of a kingdom and manage four resources: the church, the population, the military, and the economy of the kingdom. As **Figure 1.4** below shows, the four visuals at the top of each card, each representing one of the four aforementioned resources, maintain simple designs that are very easy to understand and also clearly show how much of each resource the player has depending on how "full" the icons are. This extremely simple but informative visual design maintains the awareness of the state of the game that the player needs to make informed decisions.

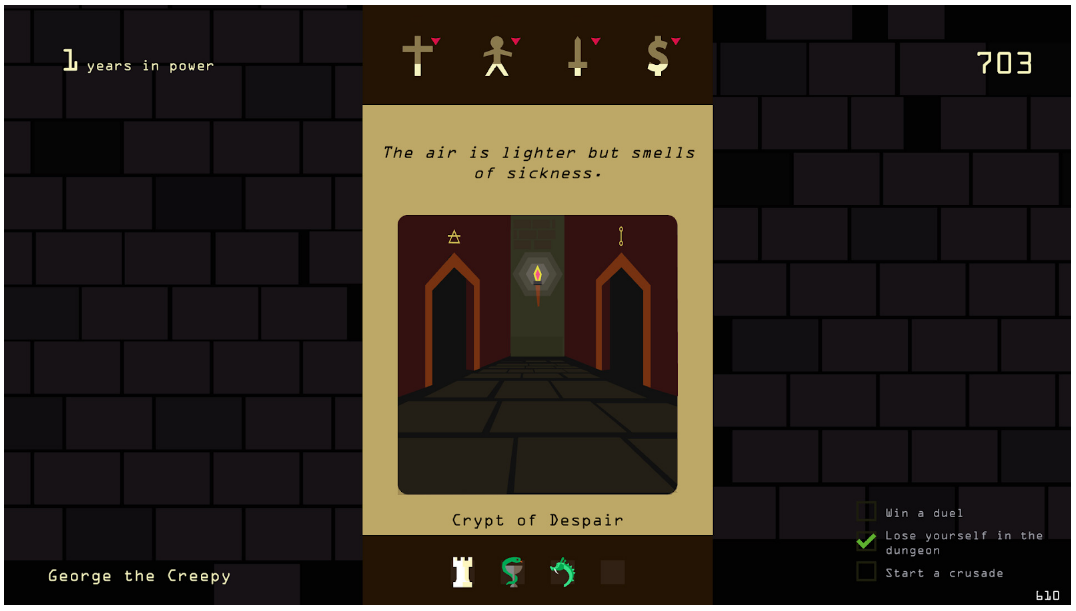


Figure 1.4: *One of the many decisions to make in Reigns, highlighting the simplistic but clear visual design [5].*

When brainstorming how our project could contribute to the space of visual design in programs, we wondered what would happen if we took the basic premise of a resource-management game such as Reigns with communicative visuals and expanded upon it in a coding context with multiple facets to manage like in Gibber. This line of thinking led to the development of our coding-based resource-management game, *Coding Through the Ages*. The main goal behind this game was to observe if the presence of visual annotations within the game would give the player a better understanding of the current state of the game and thereby give the player a better chance to complete the game successfully. Designing the game within the space of programming led us to make the visual annotations inline, so the player would have to type lines of code to actually see the visuals and progress through the game.

The planning and development of *Coding Through the Ages* was a complicated process with several different steps, including careful brainstorming of different game concepts and mechanics, detailed screen mockups and layout designs, the actual development of the game, and testing with different users and observing their gameplay data. However we even started this long sequence of events, we first had to conduct some research into different areas that would affect how we designed the game.

Chapter 2

Background

2.1 Understanding Programming

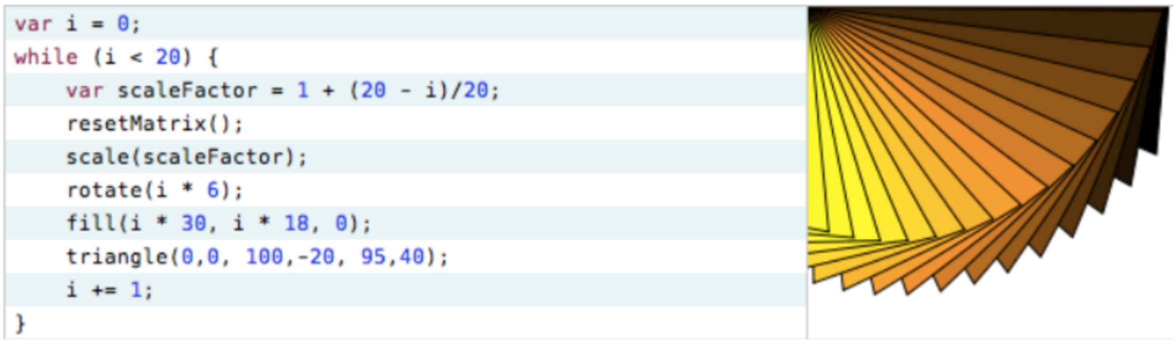
Programming has become a very selective skill as technology has advanced over the last few decades. Although more people have grown accustomed to using technology, especially computers and smartphones, in their everyday lives, the programmers who actually make the technology work need to learn to think in a completely different way. This usually means having to learn multiple different programming languages, including use cases (object-oriented vs. scripting) and different syntax-related issues, just as one would when learning a brand new language. This process may take years of exposure to different programs and may be difficult to pick up on for some.

Programs should be much more intuitive and easy to understand. Anyone with even a little understanding of technology should be able to observe a program and interpret its execution with ease. This will in turn make it much easier for programmers to come up with creative ways of solving problems through code, since they will be less focused on exact syntax. Developers of software or programming languages need to take into account the user's perspective while also making the system robust in a way that encourages powerful thinking. An effective way of designing these tools is to split it into two main components: the programming environment and the programming language [3].

The programming environment is the software that displays the code on the computer screen. As the environment is the way programmers visualize the program and its output, it should accomplish a number of goals [3]:

- Explain the meaning of the program with little effort from the programmer
- Display the flow of the program and show how it changes over time
- Reveal each state of the computer and show all data
- Encourage the programmer to translate any ideas directly into code
- Allow code to be easily expandable, applying it to more specific use cases

Meeting all of the aforementioned goals of a programming environment allows programmers to code at their highest potential. A programming environment is meant to be the canvas for programmers to use to problem-solve in creative and efficient ways. Therefore, it should not just have a lot of useful features, but it also should present those features in meaningful ways and be fully transparent with program execution. **Figure 2.1** shows a simple program and its output and details how the program should be fully transparent to the programmer [3].



```
var i = 0;
while (i < 20) {
    var scaleFactor = 1 + (20 - i)/20;
    resetMatrix();
    scale(scaleFactor);
    rotate(i * 6);
    fill(i * 30, i * 18, 0);
    triangle(0,0, 100,-20, 95,40);
    i += 1;
}
```

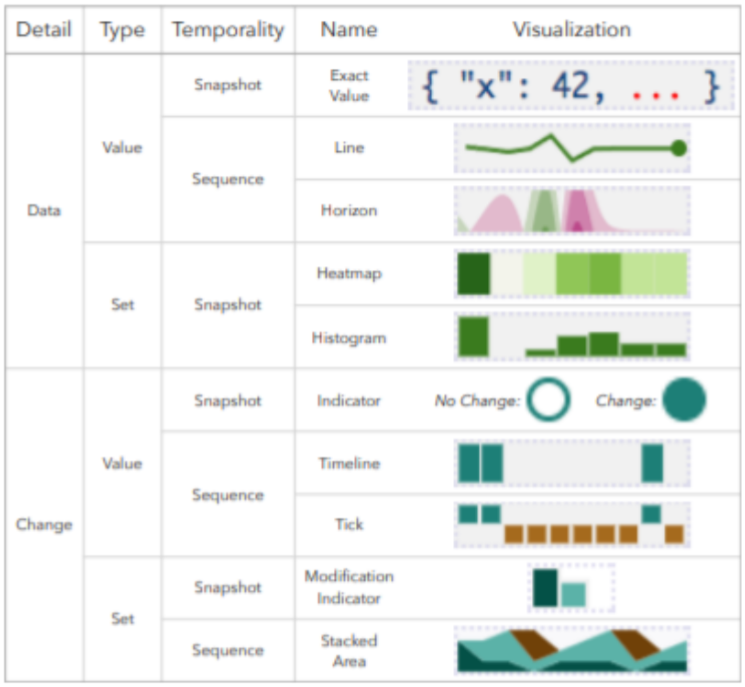
Figure 2.1: *A simple program that shows the output of each scaling and rotation on the shape on the right [3].*

The other major component of a programming system is the programming language itself. The programming language is what the programmer uses as a template to turn ideas in their head into code in the programming environment. The syntax is different between all languages, but the programmer should be able to effortlessly translate their thinking into code without needing to worry about what language they are using. Some of the goals that well-designed language should meet include [3]:

- Connect the programmer to the programming system in a meaningful way
- Allow the user to break down complex problems into simple function calls
- Facilitate the combination of different pieces of code into one program
- Explain different commands and functions through easily-interpretable syntax

These critical conditions to meet, along with those of the programming environment, will facilitate the most creative thinking possible from a programmer. The environment and language together should focus on visualizing the data rather than the code, allowing the programmer to think more about how their code actually manipulates data rather than the exact syntax and structure of the code. In addition, having the visualizations dynamically change as the programmer edits the code will make the functionality of the program even more transparent [3].

A specific type of visualization, called ‘in situ visualizations’ [6], are especially useful in showing the state of the program and the exact data being manipulated. In situ visualizations are located in or adjacent to the lines of code in a program and serve to explain the program at each line. Not only do they provide information about the contents of the program at each line, but they also maintain the flow of programming. In other words, there is no need to switch between views to debug code and discover what the program does because the in situ visualizations show exactly that information. The most effective in situ visualizations take advantage of programmer flow by providing contextual information when needed and not distracting the programmer from program execution. **Figure 2.2** shows some examples of in situ visualizations.



Detail	Type	Temporality	Name	Visualization
Data	Value	Snapshot	Exact Value	{ "x": 42, ... }
		Sequence	Line	
			Horizon	
	Set	Snapshot	Heatmap	
			Histogram	
Change	Value	Snapshot	Indicator	*No Change:* ○ *Change:* ●
		Sequence	Timeline	
			Tick	
	Set	Snapshot	Modification Indicator	
		Sequence	Stacked Area	

Figure 2.2: *Multiple examples of different in situ visualizations [6].*

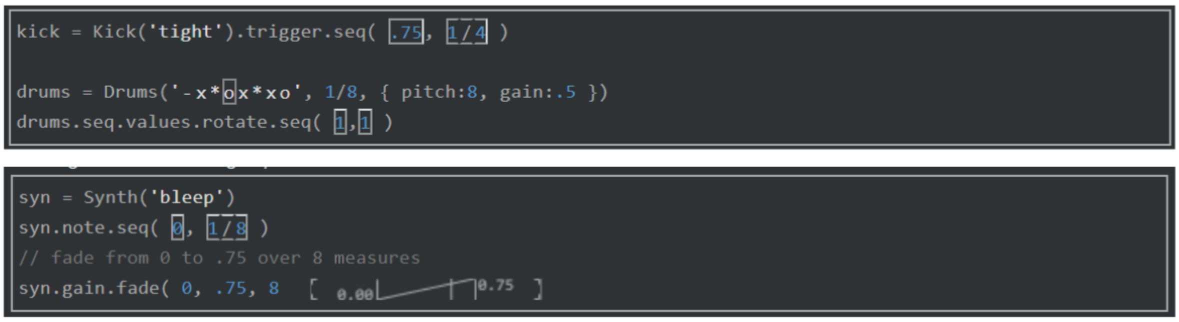
Different in situ visualizations are meant to serve different purposes. Certain visualizations, such as simple indicators or numbers, are great for showing specific values at a certain point in the program, while others, like truncated line charts or histograms, can represent a set of different data points. These visualizations can also either explain the exact data at a certain line in the program, while others, like indicators and time series plots, emphasize the change in the data over the course of the program. Every visualization can serve a purpose in making the program transparent to the programmer and keeping the programmer focused on the state of the code [6].

In their experiments, Jane Hoffswell, Arvind Satyanarayan, and Jeffrey Heer measured the helpfulness, interpretability, and intrusiveness of different types of in situ visualizations. They found that most visualizations improved both speed and accuracy of program understanding, and there was less of a disconnect between the programmer and the program due to the instant changes in the code being visible immediately. Some of the visualizations were confusing to participants, especially those relating to complex data analysis, but overall they increased the programmer's understanding of the code and ultimately maintained program flow [6].

2.2 Annotations

Annotations are defined by Merriam-Webster as a note added by way of comment or explanation. Typically used in figures and diagrams to clarify the presented information, annotations are not something often come across in programming. Live coding performances, which are digital art events that mix coding, synthesized music, and performing, showcase the utility of real-time annotations and visualizations. Such techniques mentioned in Roberts' essay Annotations and Visualizations [4] empower both the performers and the audience members, allowing them to gain greater insight into discrete events, continuous signals, and the algorithmic transformation of musical patterns.

There are many ways to display the states of variables in a program even though the viewing area is confined by screen layout. One such method is to display all updates concurrently so the audience members can choose where in the program to look at as code is developed. An advantage to this is that it minimizes the time it takes to spend on user interactions for a state. Another method - more of a principle - that Roberts mentions is similar to Gestalt theory, where an annotation must have correct closeness to display its juxtaposition. In other words, an annotation should be close enough to its corresponding information to avoid being interpreted as the explanation of some other piece of information. As **Figure 2.3** shows below, annotations can be as simple as highlighting events to represent patterns (top) or more complex visualizations depicting fades or waves for continuous data (bottom) [4].

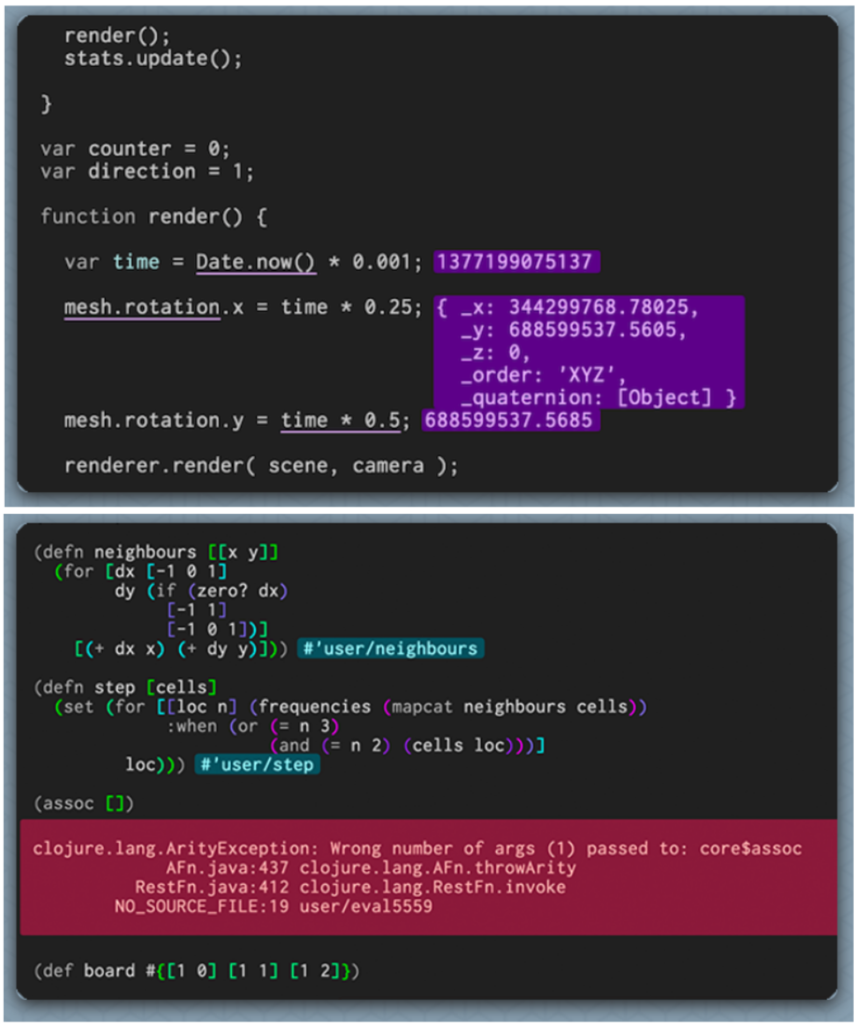


```
kick = Kick('tight').trigger.seq( .75, 1/4 )

drums = Drums('-x*ox*xo', 1/8, { pitch:8, gain:.5 })
drums.seq.values.rotate.seq( 1,1 )
```

```
syn = Synth('bleep')
syn.note.seq( 0, 1/8 )
// fade from 0 to .75 over 8 measures
syn.gain.fade( 0, .75, 8
```

Figure 2.3: *Visual representation of sounds as flashing boxes [4].*

While these visualizations are great for live coding performances, how would they translate to your typical programmer sitting in a workspace or cubicle? Programming is typically a blind job that relies on prior knowledge and experience. Why should professionals have to spend extra time just to see the result of a small change in the code? Simply put, annotations are a great quality of life improvement for all programmers. Light Table [7] is a prime example of this novelty. A code editor with built-in annotations give the programmer far more consistency when building their application. Visualizations in this type of environment are slightly different than the annotations in a live coding environment, though the concept of displaying the state of variables is the same. **Figure 2.4** below shows a couple of examples of some annotations in Light Table.



```
  render();
  stats.update();

}

var counter = 0;
var direction = 1;

function render() {

  var time = Date.now() * 0.001; 1377199075137

  mesh.rotation.x = time * 0.25; { _x: 344299768.78025,
                                   _y: 688599537.5605,
                                   _z: 0,
                                   _order: 'XYZ',
                                   _quaternion: [Object] }
  mesh.rotation.y = time * 0.5; 688599537.5685

  renderer.render( scene, camera );
```

```
(defn neighbours [[x y]]
  (for [dx [-1 0 1]
        dy (if (zero? dx)
             [-1 1]
             [-1 0 1])]
    [(+ dx x) (+ dy y)])) #'user/neighbours

(defn step [cells]
  (set (for [[loc n] (frequencies (mapcat neighbours cells))
             :when (or (= n 3)
                       (and (= n 2) (cells loc)))]
         loc))) #'user/step

(assoc [])

clojure.lang.ArityException: Wrong number of args (1) passed to: core$assoc
             AFn.java:437 clojure.lang.AFn.throwArity
          RestFn.java:412 clojure.lang.RestFn.invoke
        NO_SOURCE_FILE:19 user/eval5559

(def board #{[1 0] [1 1] [1 2]})
```

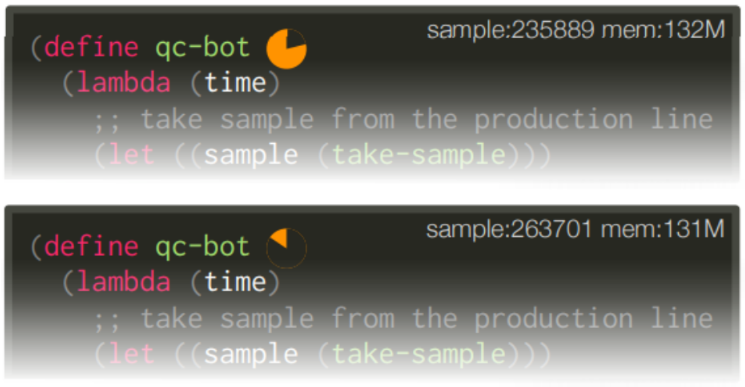
Figure 2.4: *LightTable's inline annotations [7].*

2.3 Real-time Feedback

Cyberphysical programming [8] is a type of programming in which changes to the code can happen at any moment in real time. A sub-domain of cyberphysical programming is livecoding, where a programmer write code in front of the audience in real time to create audio and visual appeal to the audience. In a cyberphysical programming environment, the code is changed and re-evaluated by the programmer's intervention. The state of the world, or the data stored in memory and the call stack, essentially describes the general environment of the code. On the other hand, the state of the code is the actual code of the program itself. Unfortunately, such a cyberphysical environment can cause the state of the world and the state of code to diverge from each other. When the state of code and state of world diverge, mistakes can happen frequently, and the programmer is likely to become confused.

In order to make the information clearer, one can implement visual annotations to the programming interface, providing the programmer with valuable information and resources, and clear out any confusions caused by the states diverging. Common types of annotations can be syntax highlighting, or visualizing current ram usage, cpu usage using graphs, and even more conceptual information such as code smells or code coverage. These annotations can be characterized into 3 categories: visualization of the state of world, state of code, or visualizing the connection between both the state of world and state of code [8].

The visualization for the state of the world is fairly straightforward. It could be anything from the performance of the code, callback rate, or other information that can help the programmer to make adjustments based on the information gained. For example, if the programmer learned that the CPU usage is rising quickly, he might as well tweak the code to remove the CPU intensive code and replace them with a similar, but better piece of code. More easy to understand graphs also exist - as seen in the Impromptu head up display demonstration (**Figure 2.5**), a circle timer is drawn near the start of code to indicate how long it will take for the code to execute again [8].



```
(define qc-bot                        sample:235889 mem:132M
  (lambda (time)
    ;; take sample from the production line
    (let ((sample (take-sample)))
```

```
(define qc-bot                        sample:263701 mem:131M
  (lambda (time)
    ;; take sample from the production line
    (let ((sample (take-sample)))
```

Figure 2.5: *The timer icon in the Impromptu heads-up display [8].*

The state of code visualization is a type of visualization directly related to the code written. Syntax highlighting, or checking for a code's syntax errors all fall in this category. By accentuating these information, the development environment can prevent unwanted waste of resources or crashes during the performance, or avoiding making a mistake when multiple loops are present. A more direct and effective way of fixing the problem also exists In the impromptu head up display demo: the code reverts back to a previous state when the speaker tried to evaluate a line of broken code, so no problems will occur in the execution [8].

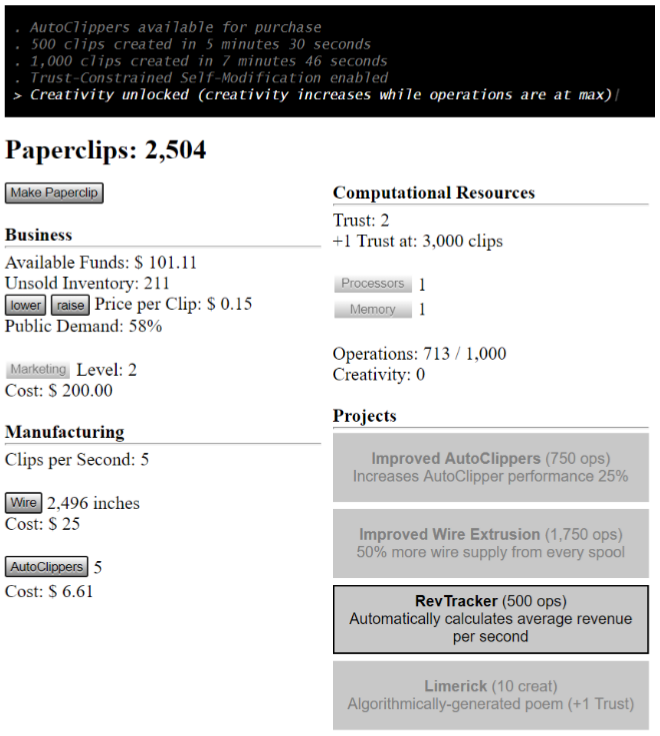
The visualization of the state of world to state of code relationship can be as simple as whether a code has been re-evaluated since the last change, or whether the code is currently executing. Throughout this kind of annotation, the programmer can have an idea of whether the two parts are diverging from each other or not. It could also be visualizing how much stress the code will put onto the system if evaluated, allowing the programmer to react to it dynamically [8].

2.4 Simulation and Resource-Management Games

Many aspects of our game were inspired by simulation, resource management, and interactive fiction games. A simulation game can be defined as a game that mimics certain aspects of real life. Typically, such games use "ticks", a set period of time to update the game. Interactive fiction games can be considered a subgenre of simulation games where players use text commands to influence the environment or control characters. Resource management games, on the other hand, involve numerous types of available materials or assets that the player need to cope with in order to win the game, usually by establishing their relative values in the gameplay context.

2.4.1 Universal Paperclips

Our team resonated with Universal Paperclip's [2] aspect of starting small and building up complexity. Universal Paperclip's game is a simulation game, in which as the player progresses, some resource management elements are introduced to make the game more difficult. Its starting interface is extremely simple, teaching the player the idea behind the game. Our team resonated with Universal Paperclip's aspect of starting small and building up complexity. Universal Paperclip's is a simulation game, in which as the player progresses, some resource management elements are introduced to make the game more difficult. Its starting interface is extremely simple, teaching the player the idea behind the game. Universal Paperclips gradually introduces many elements along it's storyline, so much that it almost fills the entire webpage, as **Figure 2.6** shows.



```
. AutoClippers available for purchase
. 500 clips created in 5 minutes 30 seconds
. 1,000 clips created in 7 minutes 46 seconds
. Trust-Constrained Self-Modification enabled
> Creativity unlocked (creativity increases while operations are at max)
```

Paperclips: 2,504

Business

Available Funds: $ 101.11
Unsold Inventory: 211
lower raise Price per Clip: $ 0.15
Public Demand: 58%

Marketing Level: 2
Cost: $ 200.00

Manufacturing

Clips per Second: 5

Wire 2,496 inches
Cost: $ 25

AutoClippers 5
Cost: $ 6.61

Computational Resources

Trust: 2
+1 Trust at: 3,000 clips

Processors 1
Memory 1

Operations: 713 / 1,000
Creativity: 0

Projects

Improved AutoClippers (750 ops)
Increases AutoClipper performance 25%

Improved Wire Extrusion (1,750 ops)
50% more wire supply from every spool

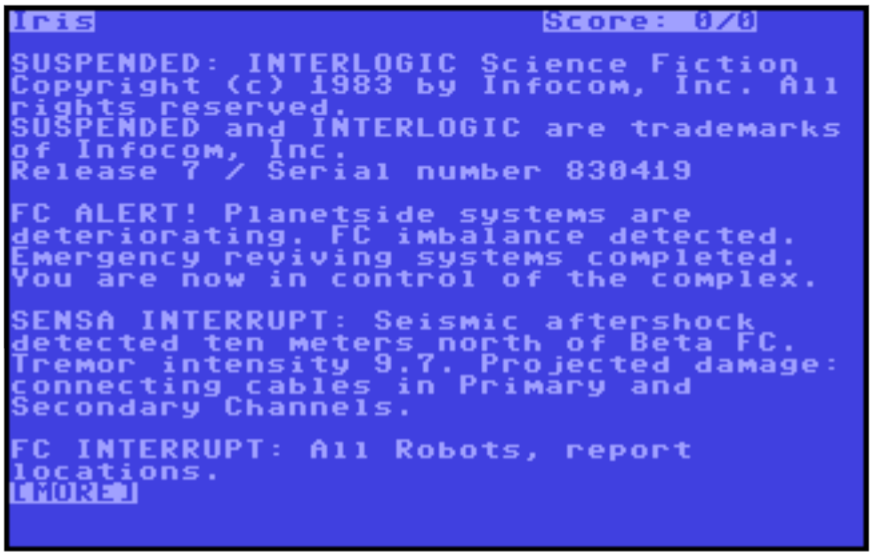
RevTracker (500 ops)
Automatically calculates average revenue per second

Limerick (10 creat)
Algorithmically-generated poem (+1 Trust)

Figure 2.6: *The Paperclip game gets more complicated with multiple different components to manage over time [2].*

2.4.2 Suspended

Suspended [9] is a classic 80's interactive fiction game. At the time, the game was a text adventure breakthrough and evolved the interactive computer novel industry. The game was bundled with extra physical items, such as a map and overview booklet, to help players keep track of their environment. In our game we wanted to experiment with visually restricting the user to a code editor-like context. We noted how complicated Suspended appeared to be without the proper equipment, and wanted to make sure our game is playable without extra tools. **Figure 2.7** shows a screenshot of the gameplay in Suspended.



```
Iris                                    Score: 0/0

SUSPENDED: INTERLOGIC Science Fiction
Copyright (c) 1983 by Infocom, Inc. All
rights reserved.
SUSPENDED and INTERLOGIC are trademarks
of Infocom, Inc.
Release 7 / Serial number 830419

FC ALERT! Planetside systems are
deteriorating. FC imbalance detected.
Emergency reviving systems completed.
You are now in control of the complex.

SENSA INTERRUPT: Seismic aftershock
detected ten meters north of Beta FC.
Tremor intensity 9.7. Projected damage:
connecting cables in Primary and
Secondary Channels.

FC INTERRUPT: All Robots, report
locations.
[MORE]
```

Figure 2.7: *The interactive console window of Suspended [9].*

2.4.3 Reigns

Our team also took heavy inspiration from Reigns [5], a uniquely simple resource management game, where the player must make choices balancing their kingdom's assets. Reigns has a very simple interface: players can swipe the card on screen to the left or right to make different decisions that affect four pools of resources. Our team wanted to emulate something very similar, and ended up basing our general concept on this game. To keep our game from getting too complicated, we decided to restrict the player's choices to two, similar to Reigns. Unlike Reigns however, our game has a linear story and does not have branching events at this time. **Figure 2.8** shows the player making a decision in Reigns.

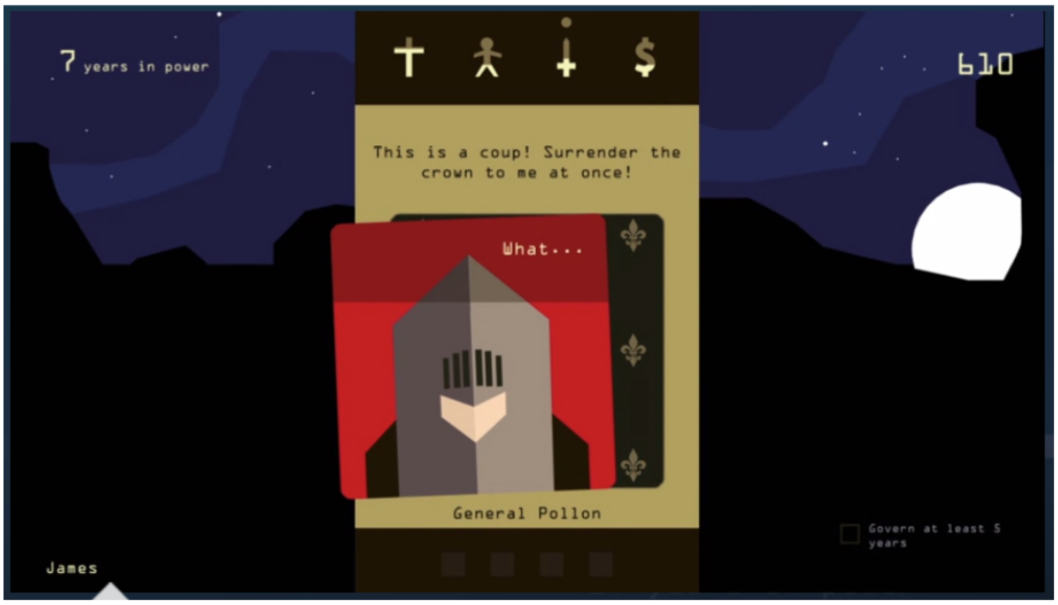


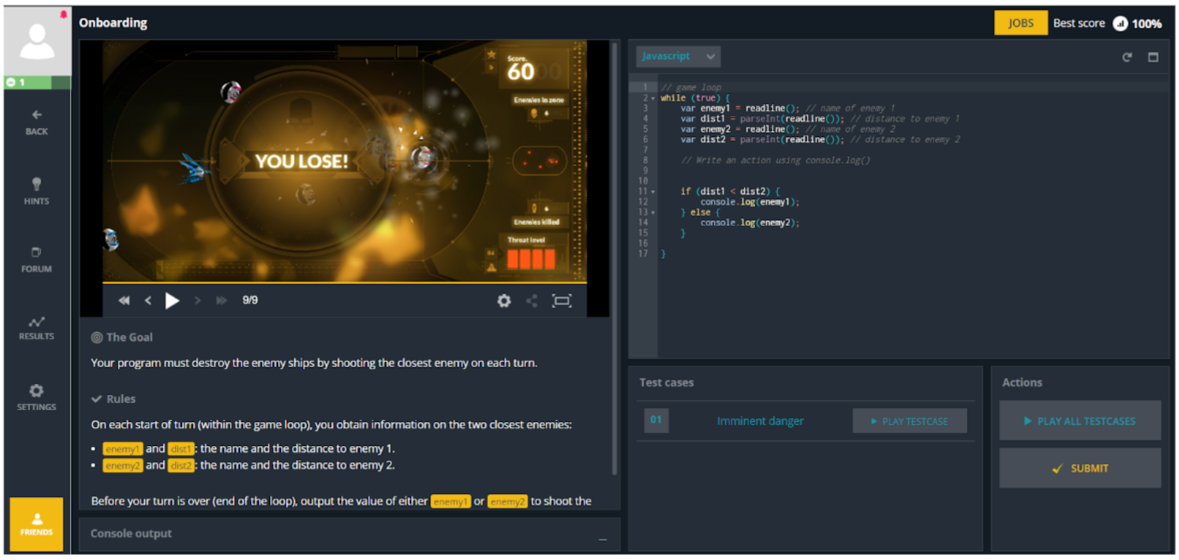
Figure 2.8: *The interface for making decisions and changing statistics in Reigns [5].*

2.5 Educational Coding Games

2.5.1 CodinGame

CodinGame [10] is an interactive coding platform that teaches users how to code with different miniature games. There are many different languages available to learn, including JavaScript, C#, Java, and Python. The user can choose what language to learn and interact with upon starting the tutorial.

Once a language is chosen, the user can begin playing the different games available. In each game, the user must add code based on the given the specified framework of the game's code. This can consist of writing for loops that iterate upon multiple game frames, designing if statements to check that certain game states are currently active, and returning values of game variables in the console log. **Figure 2.9** is taken from the tutorial game, which simply involves copying and pasting a certain block of code to get the user acquainted with the coding system [10].



```
// game loop
while (true) {
    var enemy1 = readline(); // name of enemy 1
    var dist1 = parseInt(readline()); // distance to enemy 1
    var enemy2 = readline(); // name of enemy 2
    var dist2 = parseInt(readline()); // distance to enemy 2

    // Write an action using console.log()


    if (dist1 < dist2) {
        console.log(enemy1);
    } else {
        console.log(enemy2);
    }

}
```

The Goal

Your program must destroy the enemy ships by shooting the closest enemy on each turn.

Rules

On each start of turn (within the game loop), you obtain information on the two closest enemies:

- enemy1 and dist1: the name and the distance to enemy 1.
- enemy2 and dist2: the name and the distance to enemy 2.

Before your turn is over (end of the loop), output the value of either enemy1 or enemy2 to shoot the

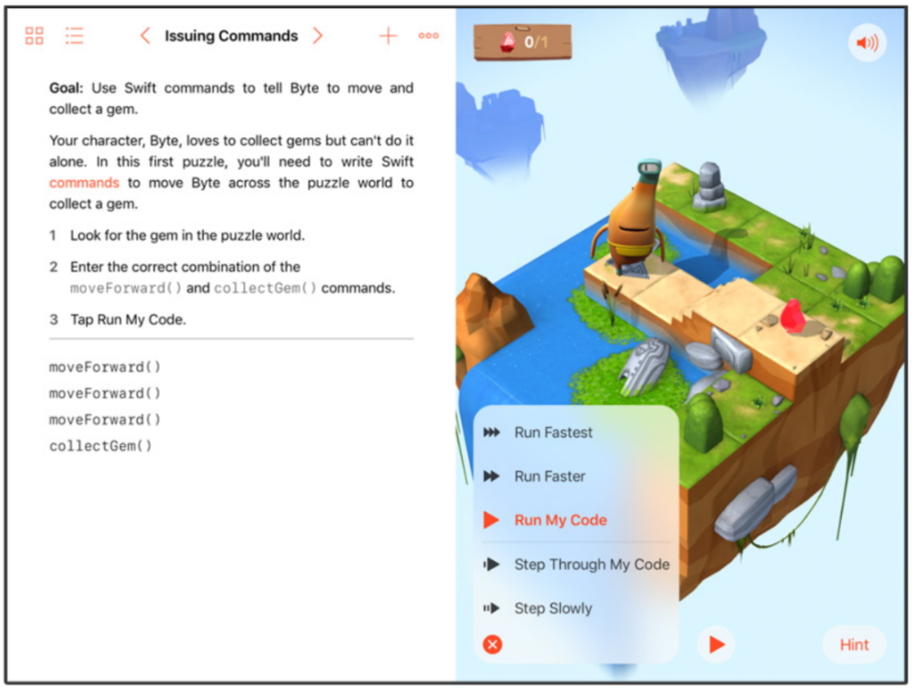
Figure 2.9: *One of the first JavaScript coding tutorials in CodinGame [10].*

Each game has at least one test to complete. Each test contains different in-game criteria to be met, and all together they test the ability of the user's generated code to work in all cases. Even if the user does not pass all tests, they can decide to submit their code and be graded based on the performance of their code on the tests [10].

The main purpose of CodinGame is to teach users how to code in a specific language while keeping the coding process fun. Each game snippet teaches users to observe the current coding framework and add code based on the current context of the program, which happens to be a game with this system. This incentivizes the user to see the game working properly by correcting their code and making sure it passes all of the tests [10].

2.5.2 Swift Playgrounds

Swift Playground [11] is an app running on both macOS and on iPadOS that allows users to attempts to teach users how to program by providing an incentive to coding: playing a game. The app user can control the movement and other actions of the main character by typing code. The player can perform certain actions, such as moving forward a space or collecting a gem, by interacting with a text editor that exists on one half of the screen. The other half contains the game area where the player can see the visualization of their code, as **Figure 2.10** below shows.



Goal: Use Swift commands to tell Byte to move and collect a gem.

Your character, Byte, loves to collect gems but can't do it alone. In this first puzzle, you'll need to write Swift commands to move Byte across the puzzle world to collect a gem.

1 Look for the gem in the puzzle world.

2 Enter the correct combination of the `moveForward()` and `collectGem()` commands.

3 Tap Run My Code.

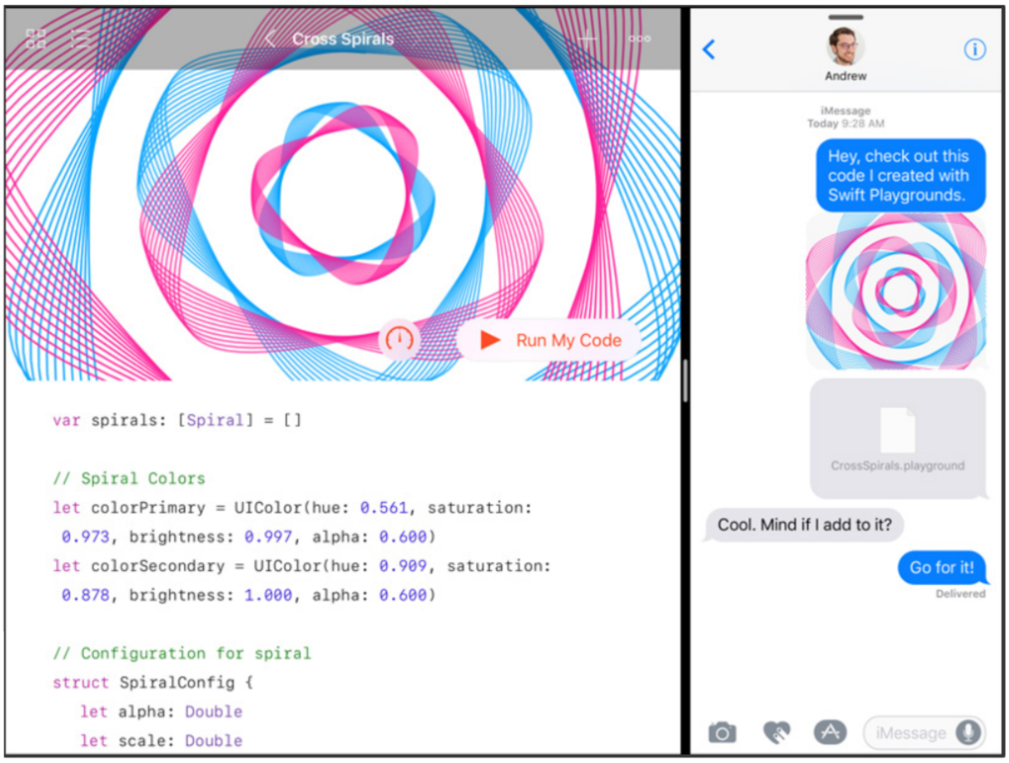
```
moveForward()
moveForward()
moveForward()
collectGem()
```

Figure 2.10: *The simple gem-collecting game Swift Playground offers in its tutorials [11].*

The code shown on the left is in a language called Swift. Swift is an Apple-created language that is used for building iOS apps. It is open source, which allows programmers to have many creative freedoms in terms of what to build, as long as it is on Apple iOS devices. Some examples of apps that were built using Swift include Lyft, LinkedIn, and Khan Academy [11].

Swift Playground uses the simple API of Swift and combines it with a gaming interface to try to teach players basics about programming. Having both the code and the game itself displayed on the same screen allows the user to quickly observe the effects that the code they typed has on the game. After the players practice basic function calls such as moveForward() and collectGem() to progress through a basic level in the game, the app introduces common coding practices such as for loops to execute the same blocks of code a certain number of times. This learning approach slowly introduces the user to programming techniques that will allow them to slowly but surely understand the essentials of coding [11].

Swift Playgrounds offers many different tutorials for that specific game, but there are other options as well, such as the visual spirals shown in **Figure 2.11** below. Users can write their own Swift code and share it with their contacts who also have iOS devices, which allows for pair-programming as well, a common trend in the coding industry. All of these accessible options allows for a lot of creative potential in Swift code and the programs that result [11].



```
var spirals: [Spiral] = []

// Spiral Colors
let colorPrimary = UIColor(hue: 0.561, saturation:
 0.973, brightness: 0.997, alpha: 0.600)
let colorSecondary = UIColor(hue: 0.909, saturation:
 0.878, brightness: 1.000, alpha: 0.600)

// Configuration for spiral
struct SpiralConfig {
   let alpha: Double
   let scale: Double
```

Hey, check out this code I created with Swift Playgrounds.

Cool. Mind if I add to it?

Go for it!

Figure 2.11: *An example of using Swift Playground to experiment with visual design [11].*

The major downside to Swift Playgrounds is that it is exclusive to Apple products, specifically the iPad. Since the iPad is a touchscreen device, some users may find it less intuitive to create code, especially users who are already experienced programmers and would find the user of a physical keyboard more efficient for typing code. The main issue with Apple exclusivity however is the smaller range of users that can work with the app. While it is true that many people do own Apple products, and many of those people own iPads, the app would gain much more traction if it was a browser game or hosted on a PC, and therefore it would gain more attention and educate more people about coding [11].

2.5.3 Kodu Game Lab

Kodu Game Lab [12] is another environment for teaching programming that targets a much younger audience. It allows children to create games for either PC or Xbox using a special visual programming language that is much different than the typical language and IDE environment developers are used to. As shown in **Figure 2.12** below, the game is run by a linkage of two actions: “When” and “Do.” Users can select certain conditions for “When” a specific action should be executed in the game, and the action to be executed can be placed under the “Do” section. This functionality emulates the idea of if-then statements in standard programming languages, but allows for a much more child-friendly interface.

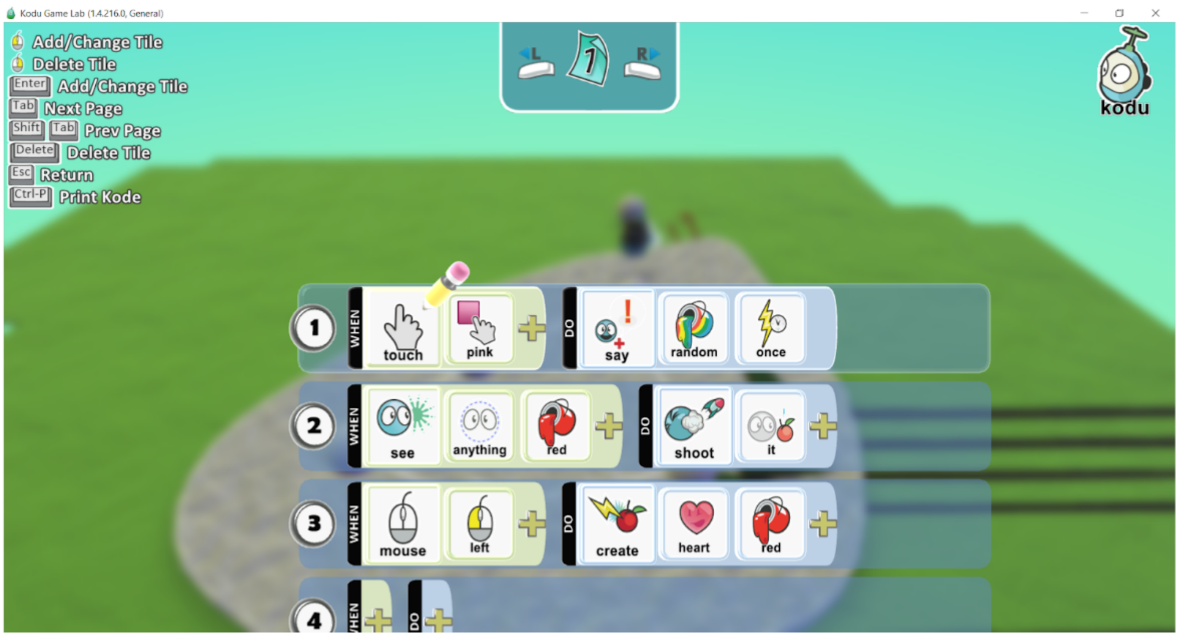

Kodu Game Lab (1.4.216.0, General)
Add/Change Tile
Delete Tile
Enter Add/Change Tile
Tab Next Page
Shift Tab Prev Page
Delete Delete Tile
Esc Return
Ctrl-P Print Kode
kodu
1 WHEN touch pink DO say random once
2 WHEN see anything red DO shoot it
3 WHEN mouse left DO create heart red
4 WHEN DO

Figure 2.12: *The very simple visual programming UI of Kodu Game Labs [12].*

Children can use this accessible language to add interactivity to their worlds. They can select different objects from a predetermined list of models, assemble their world, and then use the language to interact with the different objects in the environment. They can then share their worlds online for anyone else to download [12].

Of course, due to the simplicity of the game, it is extremely limited in terms of what it can accomplish technically. However, this is completely intentional, as the target audience is younger children. Kodu Game Labs is meant to introduce the overall concept of programming to make a game playable at an early age. That way, when the children who use Kodu grow up, they will already have some basic understanding of coding and may be able to pick up on typical programming much more quickly [12].

Kodu Game Labs is quite outdated, and not many updates have been released in recent years. The community is still thriving, and newly-created worlds can still be downloaded from the website. It served as a great example of how to teach kids the main ideas behind coding and how it can be used to make programs function [12].

Chapter 3

Methodology

Visual feedback has been shown to mitigate the confusion a user may have about a program, whether that program is a block of code within a coding environment or a task in a computer game. Our goal with this project is to understand the correlation between the presence of visual annotations and a player's experience within an environment with both coding and gaming aspects.

To achieve this goal, we developed a coding-based game that utilized supplementary, inline visuals. We accomplished this through the following methodology:

- Brainstorming game ideas that utilized visual annotations in code.
  - Planning out the actual concept of the game.
  - Creating balanced game mechanics along with unique gameplay ideas related to visuals.
  - Idealizing a narrative structure for the game.
- Designing different potential screen layouts of the game to prioritize visual information.
  - Completing many design iterations to prioritize screen space and player attention.
  - Brainstorming more visual annotation ideas to compliment the layout.
- Implementing the game mechanics and visual elements into the game.
  - Developing a command parser within a code editor to serve as a game framework.
  - Structuring a sense of progression through an introduction and story.
  - Adding different visual elements to supplement the player's code.
  - Recording gameplay data and storing into a database for later analysis.

-

- Testing the effects of the inline game visuals for later analysis.
  - Setting up online Zoom playtesting sessions.
  - Planning the session procedures.
  - Running the sessions and collecting data from in-game, think-aloud note-taking, and a survey.

3.1 Brainstorming

The first task we accomplished after understanding our project goal was planning what we wanted our actual project deliverable to be. To relate to the topic at hand, we had to incorporate some form of visual feedback alongside code. We came up with several different ideas, including a coding game with mechanics similar to Space Invaders, a coding assistant like Clippy from Microsoft Word, and a mathematics-based annotation system. We knew early on that we wanted users to apply their programming skills and be able to see the effect of their code in some visual form.

After a followup discussion with our project advisors and some more brainstorming, we decided to create a "text adventure" type of game where players would interact with a text-based window to make decisions in response to different challenges. Similarly to the game Suspended [9], players would enter commands to respond to the different prompts they were given. As we thought of different commands that the user could enter, we took inspiration from games such as Reigns [5] and Universal Paperclips [2] and added a resource-management component. The player would need to balance six different statistics within the game: their own hunger level, the amount of food they had, the security of the computer terminal that they interacted with, a human population they had control over, the military protecting that population, and the amount of science (or knowledge) that the community had. Having all of these components to manage justified the ability to enter commands and also added complexity to the game.

Once we thought of these core mechanics, we began to plan out the complexity behind balancing all of the statistics. We introduced a time limit to increase tension and force players to make a decision within a given amount of time. These decisions would affect some of the statistics, specifically population, military, and science, but in addition, all of the statistics besides food and security would slowly decrease with time. The level of each statistics would affect the rate at which others decreased: having zero population or science would decrease population faster, zero population would decrease hunger faster, and zero hunger would result in a game over. All of these ideas made it into the final version of the game, but some other early ideas are shown in **Figure 3.1** below.

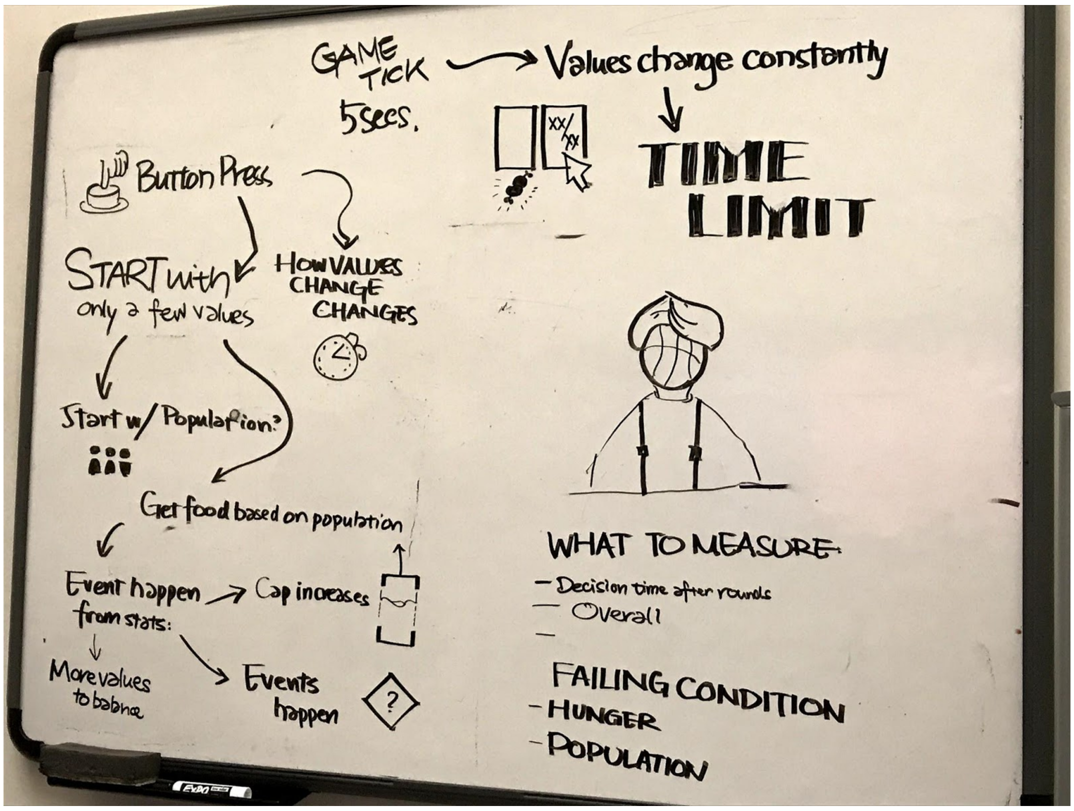


Figure 3.1: *Early ideas of some of the major mechanics for our coding-based game.*

Since this was a coding-oriented game, we needed to choose a familiar programming language that would be able to host our game and also allow players to interact with the game code via commands. Therefore, we decided to build the game in JavaScript and HTML, which would allow our game to easily be hosted in a web browser while having simple syntax rules that we wanted our target audience to understand. The entire project team also had a moderate amount of prior JavaScript experience, which made developing the game much easier and more efficient.

Early on, we had hoped to create an experimental setup where users would play different versions of our game, each with different amounts of visual annotations present, and we wanted to observe how they performed in each version. Originally, we came up with three different experimental cases: one with extremely limited visuals (base case), one with slightly more visuals (experimental case) and one with a multitude of visuals (other experimental case). The number of different cases decreased as we designed and developed the game, but we still needed to finalize our experimental measurements. Some early ideas we had for tracking player performance were measuring the time it took for a player to enter a command and using a JavaScript library to track player eye movement on certain portions of the computer screen. However, many of these ideas completely changed as we began to brainstorm how the game would actually look.

3.2 Design

As we discussed our brainstorming ideas over the first half of the school year, we created many design mockups, making sure to consider how certain elements like statistics and other visuals would appear within the same space. Screen layout was extremely important, as we wanted to create a flow between each distinct visual element for the player to form a more cohesive gameplay experience. We spent a considerable amount of time thinking about different potential layouts before making official mockups. **Figure 3.2** below shows some of the ideas we originally came up with.

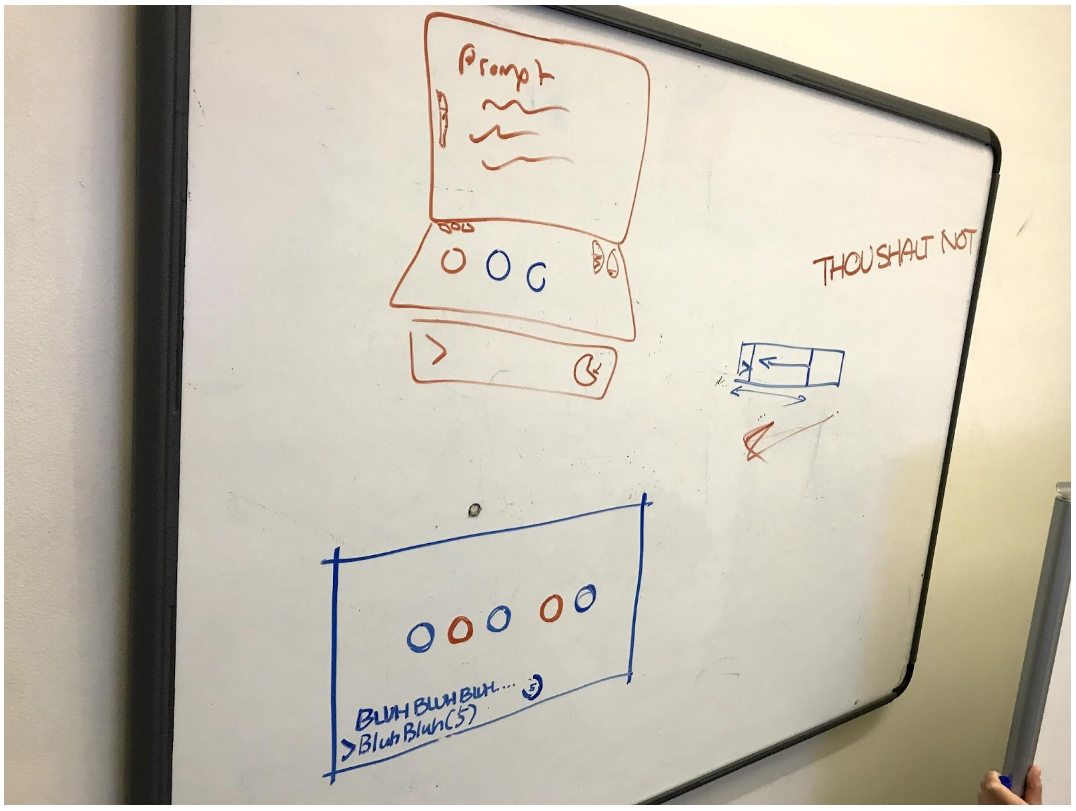
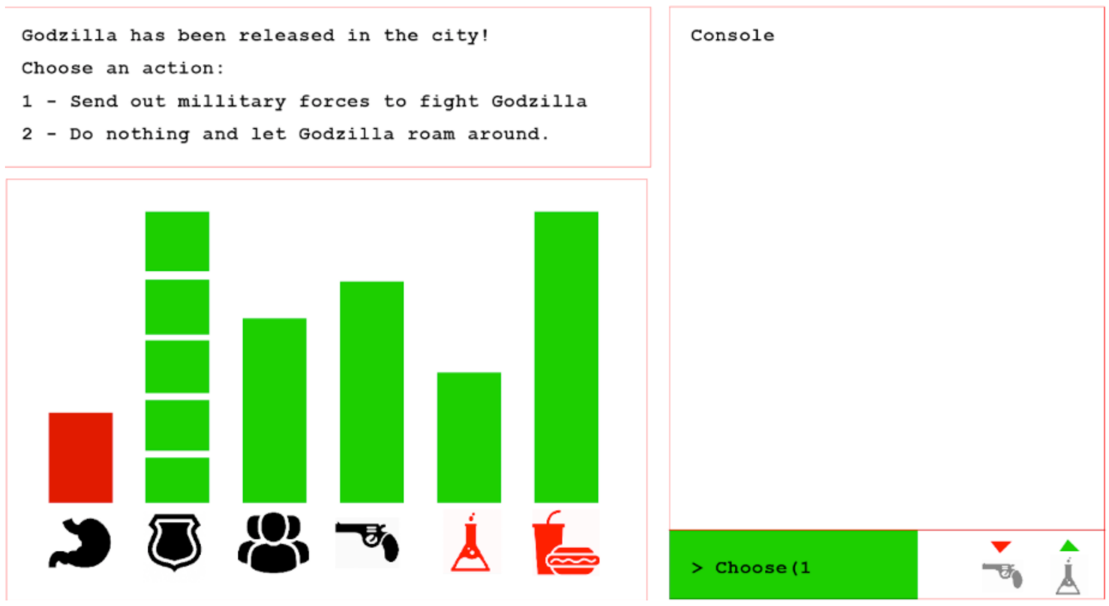


Figure 3.2: *Early ideas on arranging the elements of the coding-based game.*

Our first official design iteration conveyed many of the main features we discussed during the brainstorming phase. As seen in **Figure 3.3**, the visuals for the statistics (the vertical bars) occupied a large portion of the screen: we wanted the players to clearly see the fluctuating bars to influence their choices, but there was originally almost too much of an emphasis placed on the visuals. There were also stricter, more defined areas of the screen that each represented a different component of the game, with each being a general container that split up the viewed information.


Godzilla has been released in the city!
Choose an action:
1 - Send out millitary forces to fight Godzilla
2 - Do nothing and let Godzilla roam around.

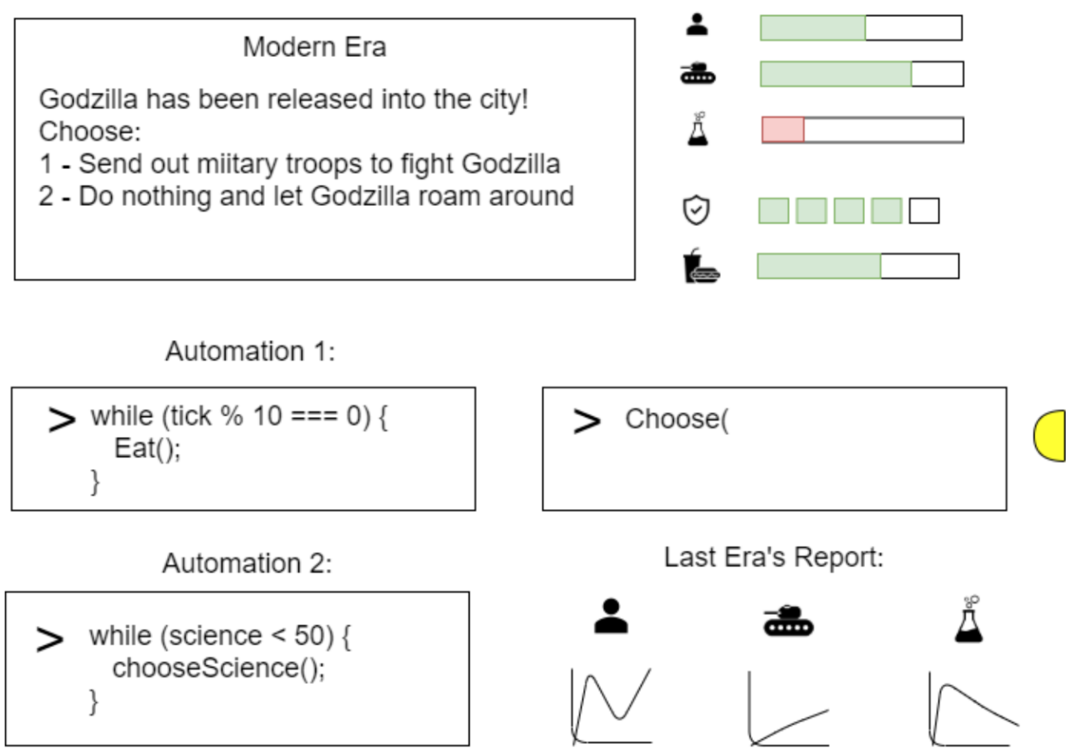
Console

> Choose(1

Figure 3.3: *First iteration, emphasizing visuals and separation of components.*

During this first iteration, we began storyboarding and coming up with different creative ideas for prompts, their choices, and the affected statistics. We started coming up with different events that the player would have to respond to, and each of these events had some influence from real life or common sci-fi tropes, such as Godzilla destroying a city and aliens invading the earth. Taking inspiration from these media definitely added some familiarity to the game and would in theory help players to make their own decisions.

Our second batch of mockups prioritized the player's eye movement within the page space. As seen in **Figure 3.4**, all of the elements were organized into separate containers. We originally wanted all elements to be visible at all times for the player's convenience, so the screen was organized to allow for that. The prompt and visuals for the current statistics were placed at the top of the page, as they represented all of the contextual information the player would need to make choices. The bottom half of the screen was dedicated to areas where the player would input their code. At this point in time, our experiment idea was to use an eye-tracking JavaScript library and track which half, the top or the bottom, the player spent most of their time looking at. We theorized that with added visuals in the bottom half of the screen, the player would not need to look at the prompts and bars in the top half too often.



Modern Era

Godzilla has been released into the city!
Choose:
1 - Send out miitary troops to fight Godzilla
2 - Do nothing and let Godzilla roam around

Automation 1:

```
> while (tick % 10 === 0) {
    Eat();
  }
```

```
> Choose(
```

Automation 2:

```
> while (science < 50) {
    chooseScience();
  }
```

Figure 3.4: *Constantly showing relevant statistics, history and multiple automation slots.*

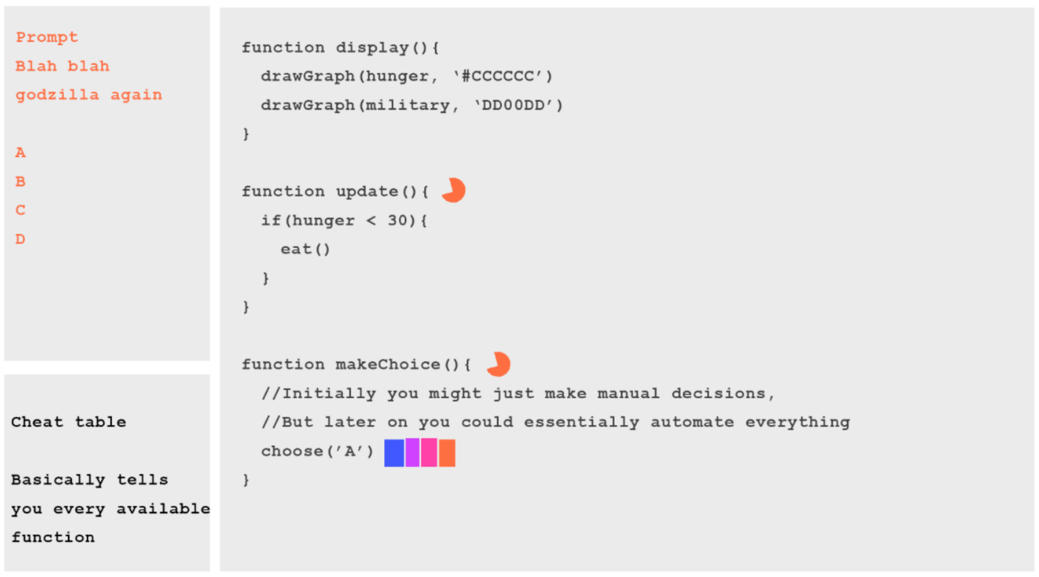
It was during this design iteration when we added two new features into the game: automated blocks of code and showing previous changes in statistics. The automated blocks of code were not fully realized, but they would essentially allow players to type any code and have it run every frame of the game. Visualizing the changes in statistics, represented by the line graphs in the figure, would cause players to prioritize certain statistics over others when making a final decision. These two new components added more strategy to the game and also used up more screen space in the bottom half.

The “statistic history” feature fully came about when we decided to split up parts of the story of our game into different eras in time. We specifically based the natural progression of events on humanity’s timeline, starting with the Stone Age and ending in a futuristic time. This organized structure helped us come up with many more ideas for prompts since we could base some of them off historical events and add familiar context to the game to appease players. This also made the user’s past decisions much more organized, since the change in statistics was simply displayed for the previous age rather than a long list of all the past choices the player made.

As we continued reorganizing the layout through many iterations, we made sure to modify the statistics visuals to take up less space as well as making it stylistically different. We changed each bar within the visual to have its own color corresponding to a different statistic, and we changed the color scheme to invoke the feeling of using an old computer terminal with green text but with additional color in the visuals.. Along with colors, we also explored more options for different features, such as having the timer be a circle that depleted in a counter-clockwise fashion and integrating more visuals into the prompt window to make the game less dull and more visually appealing.

We eventually reached a point where our advisors envisioned a more programming-oriented focus for automating tasks and potentially the entire game. We considered how much we wanted to constrain the player's ability to automate for more strategic thinking and decision making. Once we started putting emphasis on more editor-focused layouts, we could make the game have a greater focus on programming and be able to automate different processes easier.

Since we decided to have a heavier focus on programming this time, we included a small section that would act as a cheat table or guide for the player that had the predetermined commands that the player would use. We explored many options regarding the timer including clock-like, text, and boxed timer visuals. As this seemed like a lot to juggle for the player, we decided to explore functionality that would slowly introduce all the game elements to the player. **Figure 3.5** below shows a design layout that leaned further towards a singular, coding-focused view of the game with visuals, commands, and automated code all integrated into the same general area.



```
Prompt
Blah blah
godzilla again

A
B
C
D
```

```
Cheat table

Basically tells
you every available
function
```

```
function display(){
  drawGraph(hunger, '#CCCCCC')
  drawGraph(military, 'DD00DD')
}

function update(){
  if(hunger < 30){
    eat()
  }
}

function makeChoice(){
  //Initially you might just make manual decisions,
  //But later on you could essentially automate everything
  choose('A')
}
```

Figure 3.5: *An early design of a pure coding editor layout.*

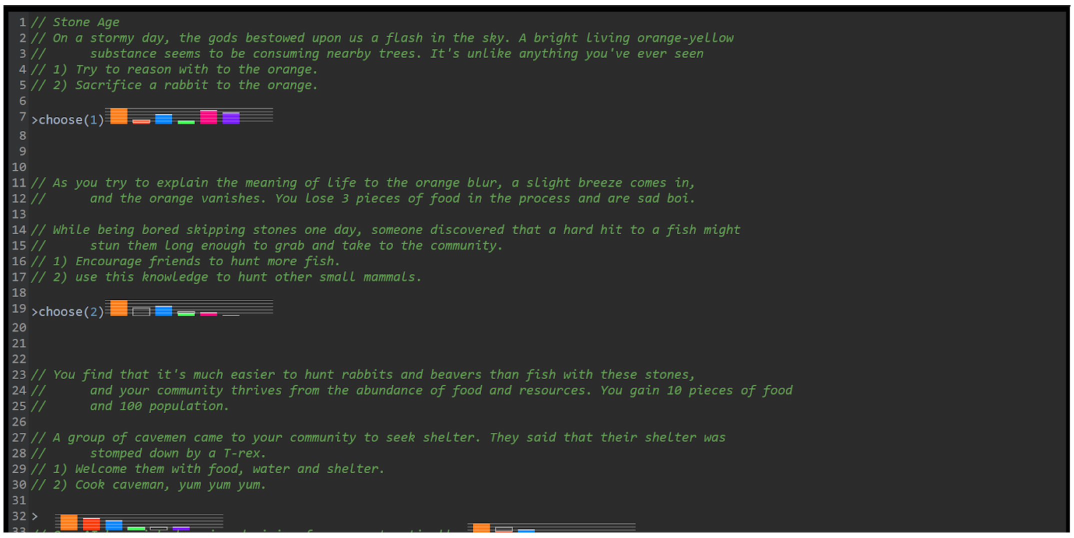
Once we confirmed with our advisors that we were definitely focusing more on the coding environment style of the game, we kept reiterating on design ideas until we consolidated all of the components, prompts and code together, into a single interface. It was during this last phase of designs that we began to look into ways to start actual development of the game. Although we had not completely finalized the full design of the game, we had enough information to work with and therefore started implementing features that we had brainstormed earlier on.

3.3 Implementation

After building a simple prototype to prove our idea, we started structuring the underlying code and building the core functionalities for the game. We used the JavaScript library CodeMirror [15] to add coding capabilities to our web-based game. CodeMirror creates an HTML widget in your webpage that acts as a code editor, enabling users to enter and edit code as if they were in a normal programming environment. The library provides many other benefits, such as line numbers, color formatting, and command execution.

We stored our story prompts in a JSON format, and each prompt would sequentially be displayed in the CodeMirror editor. With CodeMirror's line widget functionality, we were able to create visual annotations in the CodeMirror display using the HTML5 canvas tag. Many visual elements were adjusted along the process of development, such as coloring and text size, in order to improve aesthetics and readability. We also came up with different inline visual annotations to see which ones were easiest to comprehend within the game. By adjusting the position of the visuals based on the current cursor position, the players' eye movement would be reduced, allowing them to focus on writing code without any distractions.

As we developed the game, several design choices from earlier on had to be changed due to development restrictions, and new design ideas were formed to further enhance the game. We eventually consolidated all of the information being displayed on the screen into the CodeMirror widget instead of having separate areas or tabs within the game. **Figure 3.6** below shows a more streamlined, programming-focused design that we developed at this point.



```
// Stone Age
// On a stormy day, the gods bestowed upon us a flash in the sky. A bright living orange-yellow
//      substance seems to be consuming nearby trees. It's unlike anything you've ever seen
// 1) Try to reason with to the orange.
// 2) Sacrifice a rabbit to the orange.

>choose(1)



// As you try to explain the meaning of life to the orange blur, a slight breeze comes in,
//      and the orange vanishes. You lose 3 pieces of food in the process and are sad boi.

// While being bored skipping stones one day, someone discovered that a hard hit to a fish might
//      stun them long enough to grab and take to the community.
// 1) Encourage friends to hunt more fish.
// 2) use this knowledge to hunt other small mammals.

>choose(2)



// You find that it's much easier to hunt rabbits and beavers than fish with these stones,
//      and your community thrives from the abundance of food and resources. You gain 10 pieces of food
//      and 100 population.

// A group of cavemen came to your community to seek shelter. They said that their shelter was
//      stomped down by a T-rex.
// 1) Welcome them with food, water and shelter.
// 2) Cook caveman, yum yum yum.

>
```

Figure 3.6: *A more finalized design with all functionalities in one window.*

Working everything into one unified space meant we needed to rework some of the core pieces of functionality within the game to work within the one CodeMirror editor. This required us to add more commands, and eventually we came up with the following list of commands:

- choose(number) - make a choice in response to a story prompt, with the given number corresponding to the choice.
- eat() - eat a piece of food (if you have any food left)
- secure() - raise your security by 1 (if allowed to do so)
- man() - bring up a manual listing all of the commands you can do (or the usage of a specific command if you specify it as a parameter)
- legend() - pull up a legend mapping the color of the bar visuals to specific statistics
- automate(function) - create an automated function that will run at the current every frame if the specific condition within it holds true

The other major addition that was made during this phase was the actual inline visual designs. We went through several iterations of designs for different aspects of the game. For the statistics, for instance, we thought of having vertical bars, horizontal bars, and circles that changed over time when the corresponding statistics changed. We decided on vertical bars because they were not too small to read, were easy to interpret, and were noticeable when they changed. To add an extra visual element, when the player typed "choose(" and then a choice number before actually executing the command, the vertical bars would extend to show the effect that choice would have if the player finished the command. The timer visual was made into a horizontal line beneath the last line of the CodeMirror editor that got smaller over time. **Figure 3.7** below shows a close-up of some of the visual designs we settled on.

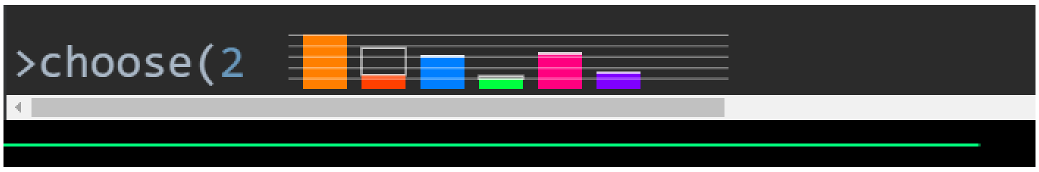


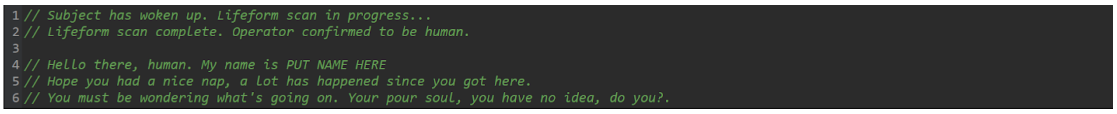
Figure 3.7: *The vertical bar and timer inline visuals within the context of the game.*

We added some new mechanics as we developed the game to provide a more interesting and balanced game experience. One of these mechanics was the security system. Although we had the idea of the security statistic early on, we were not entirely sure how it would affect the gameplay. When we got to this phase, we decided to have security factor in some randomness to keep players on their toes. The level of the security statistic would correspond to a value between 0 and 1. That value represented the probability that a choice would automatically be chosen for you after the timer reached the halfway point for that prompt. The lower the security level, the greater the chance that a random choice would be made on that prompt. The player had a chance to raise their security after 3 prompts were decided on by the player, assuming their security was not at a maximum of 5. This feature was also linked to when the timer ran out for the current prompt.

To more clearly understand the relationship between annotations and player performance, we considered adding a Firebase implementation to gather the quantitative data like the variance in statistics and the time it takes for the players to make each decision, as well as noting how many times the player takes too long to make a decision. We used a unique session id to track each player data anonymously.

As we got further in development with the game, we wanted another way to gather data, specifically feedback from the players themselves. Since we started to move towards playtesting instead of a fully fledged experiment, we anticipated these opinions would help gather more qualitative data as well as judging how much enjoyment they get out of the game. Because we were refraining from collecting personal data, to link between a player's Firebase data and their Google Survey responses, we utilized the game session's unique id for both data sets.

At this point, we discussed adding a small tutorial at the beginning of the game to get players familiar with inputting our syntax. This went through a couple of iterations, initially starting with a separate page with the relevant information and a button to go to the game. As this was essentially an information dump for the player, and did not allow the player to practice, it did not meet our expectations to be a good tutorial. The finalized version not only gives context and story elements to the player, but also enables them to experiment with our syntax to move on to gameplay. **Figure 3.8** below displays part of the tutorial screen and how it introduces the player to the core functionalities within the game.



```
// Subject has woken up. Lifeform scan in progress...
// Lifeform scan complete. Operator confirmed to be human.

// Hello there, human. My name is PUT NAME HERE
// Hope you had a nice nap, a lot has happened since you got here.
// You must be wondering what's going on. Your pour soul, you have no idea, do you?.
```

Figure 3.8: *Screenshot of part of the tutorial, which introduces basic mechanics and story.*

Another gameplay element that was added to help the player was one of the main features we discussed in our previous designs. Showing how the statistics changed over the course of each age seemed like relevant supplementary information. We anticipated this feature to further reinforce the goal of balancing each resource. Discussing the best way to show this graph led to a choice between a general trend line or a more detailed chart showing the effects of each choice. Upon some exploration in this area and advisor feedback, we actually chose neither option. Instead, we created small sparklines that were to the right of the bar visuals and still inline with the code. Each sparkline corresponded to a different statistic and captured the change in each statistic over the last five decisions. **Figure 3.9** below shows our implementation of the sparklines in the final version of the game.

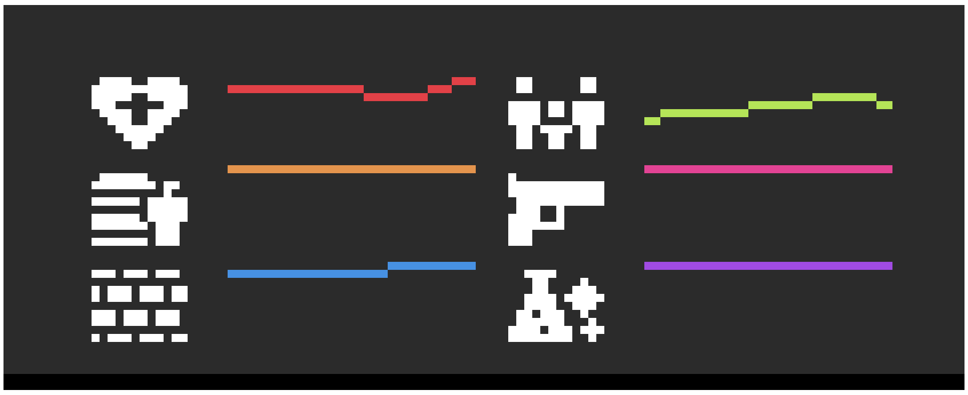
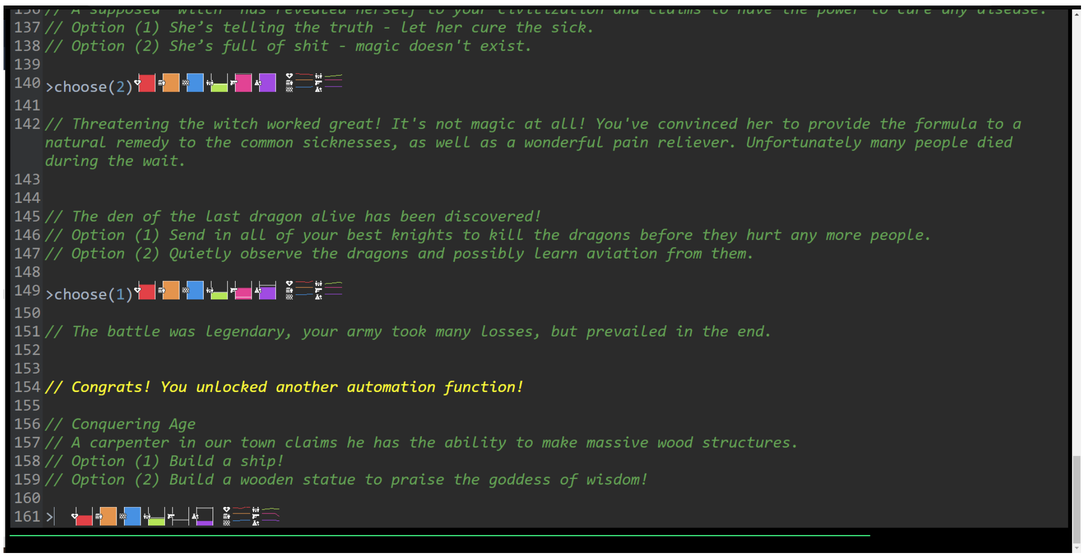


Figure 3.9: *Sparklines for each statistic shown inline next to the bar visuals.*

As we continued development of the game, we made sure to go back and refactor some of our outdated code as well as fix any bugs or issues that came up. Although we originally began developing the game before we finalized our design, we knew that we wouldn't fully know what the game would and should look like until we began implementing it via code. After giving our visuals a new 8-bit inspired aesthetic and making a few other minor changes, we considered *Coding Through the Ages*, our game, complete! **Figure 3.10** below shows a screenshot of the final form of the game, with all of our additional design ideas in place.



```
// A supposed "witch" has revealed herself to your civilization and claims to have the power to cure any disease.
// Option (1) She's telling the truth - let her cure the sick.
// Option (2) She's full of shit - magic doesn't exist.

>choose(2)

// Threatening the witch worked great! It's not magic at all! You've convinced her to provide the formula to a
natural remedy to the common sicknesses, as well as a wonderful pain reliever. Unfortunately many people died
during the wait.


// The den of the last dragon alive has been discovered!
// Option (1) Send in all of your best knights to kill the dragons before they hurt any more people.
// Option (2) Quietly observe the dragons and possibly learn aviation from them.

>choose(1)

// The battle was legendary, your army took many losses, but prevailed in the end.


// Congrats! You unlocked another automation function!

// Conquering Age
// A carpenter in our town claims he has the ability to make massive wood structures.
// Option (1) Build a ship!
// Option (2) Build a wooden statue to praise the goddess of wisdom!

>
```

Figure 3.10: *Screenshot of the final version of our game, Coding Through the Ages.*

3.4 Testing

Before COVID-19 became an international pandemic, we had planned to have in-person playtesting on the WPI campus. We intended for CS or technical IMGD majors to play our game and receive playtesting credit for their IMGD courses. This, however, needed to change once students could no longer be on campus. We began to consider our options for remote playtesting, such as crowdsourcing and online playtesting. Even though full remote options like crowdsourcing often scale better and result in a much larger audience, we were concerned with how much more work would need to be done to set up remote options such as prerequisite tests. (We conducted plenty of previous research on crowdsourcing that was originally in our Background section. See **Appendix D** for further details.)

We felt that there were more benefits with doing an online playtesting session hosted on a virtual platform, where we could still interact with our players. This required much less set up than crowdsourcing, and we could obtain immediate reactions to the visuals in our game in a live remote playtesting session. Due to the IMGD department no longer requiring playtesting credit for the final term of the year, when we planned on conducting our tests, we discussed other compensations to award participants, such as gift cards to local businesses or donations to charities. In the end, we decided on giving each player a $10 Amazon eGift card as compensation, since purchasing and distributing them was extremely easy to do remotely.

During this testing stage, our group planned for the setup, procedure, and data collection for each individual taking part in our playtesting. These were organized as described in the following subsections.

3.4.1 Setup

- During signups for playtesting sessions, players enter their name and email into a spreadsheet next to the specific day and time on which they want to play. The email is used for sending playtesting confirmation after the session and also for receiving the $10 Amazon eGift card after particpation.
- Two of the group members join a Zoom session: one person leads the session and talks directly to the player, and the other person takes notes.
- At the appropriate time, the scheduled player joins the online session.

3.4.2 Procedure

- The player reads through and electronically signs (upon agreement) a pdf copy of the informed consent form, and the session leader provides a brief overview of the game to the player.
- If they decide to stay, the player will be directed to the GitHub pages link that hosts the game, which can be found here. The game lasts approximately 20-30 minutes, and the player may restart the game as many times as they like within the time allotted for the session.
- When they finish their game session, the player completes a survey about their gameplay experience.
- The player leaves the playtesting session, and they are sent an email thanking them for their time and contribution and later another email with the $10 Amazon eGift card.

3.4.3 Data Collection

We collected the following data from each player during the course of the game in no particular order:

- The time it took for the player to make each decision
- The values of each statistic made at each decision and at the end of each age
- The number of times the player utilized the in-game manual
- The number of times the player ran out of time during a decision
- Think-aloud notes during the game session
- An anonymous, unique user ID generated by the program

- Post-game survey regarding topics about:
  - Coding ability
  - Opinions on inline visuals in game
  - Enjoyability and improvements

As development of the game continued and different features were added or changed, our testing guidelines had to be refined. We made sure to start on these procedures early enough in case of situations like the pandemic. We also had to wait a little while for the WPI IRB to approve of our playtesting proposal as well as our survey questions (**Appendix A**), our study protocols (**Appendix B**), and our informed consent form (**Appendix C**). See the corresponding appendices for more details.

Once the playtesting materials were approved and our game had been developed to a playable stage, we set up playtesting sessions for CS and IMGD majors and began collecting important results.

Chapter 4

Results

To being the playtesting process, we sent basic premise and procedure information to the WPI IMGD and CS students mailing lists. 20 students signed up for a playtesting session, although not everyone could show up. By the end of the week in which we conducted all of our playtesting, 18 players were able to make it to their sessions. These players were later sent a 10$ Amazon gift card as compensation for their participation.

After conducting all of our playtesting sessions, recording the gameplay data from multiple playthroughs of each player and storing it in a database, and collecting survey responses from each participant, we analyzed the data and visualized the results. Some of the graphs were made using the JavaScript library D3, and the others were pulled from the responses from the Google forms survey; these visualizations are accompanied by direct quotes from participants.

We first wanted to visualize a topic that is common to almost any type of game: how did the players perform? Using all the statistics from each player recorded at the end of each age (except for food, since that was dependent on hunger which we were already recording), we can observe gameplay performance in a large box-and-whiskers plot. The x-axis for the plot represents each age for each statistics, and the y-axis represents the percentage of the total bar that the statistic value was recorded at. We separated the original plot by statistic to show the individual statistic distributions over each age, but the original combined plot can be found here.

Figure 4.1 shows the distribution of hunger values over the course of each age. Overall, the distribution remains relatively the same across ages. However, it is important to note that the average recorded hunger value at the end of the first age is much higher than the average hunger values at the end of the other ages. In addition to the average recorded hunger value dropping after the first age, the number of collected data points also drops, implying that many playthroughs of the game ended after during the second age. (This is the case with all of the plots.) Finally, although the overall distribution remains roughly the same across each age, there are more outliers with each progressive age, and by the end of the final age, the recorded hunger values are quite varied.

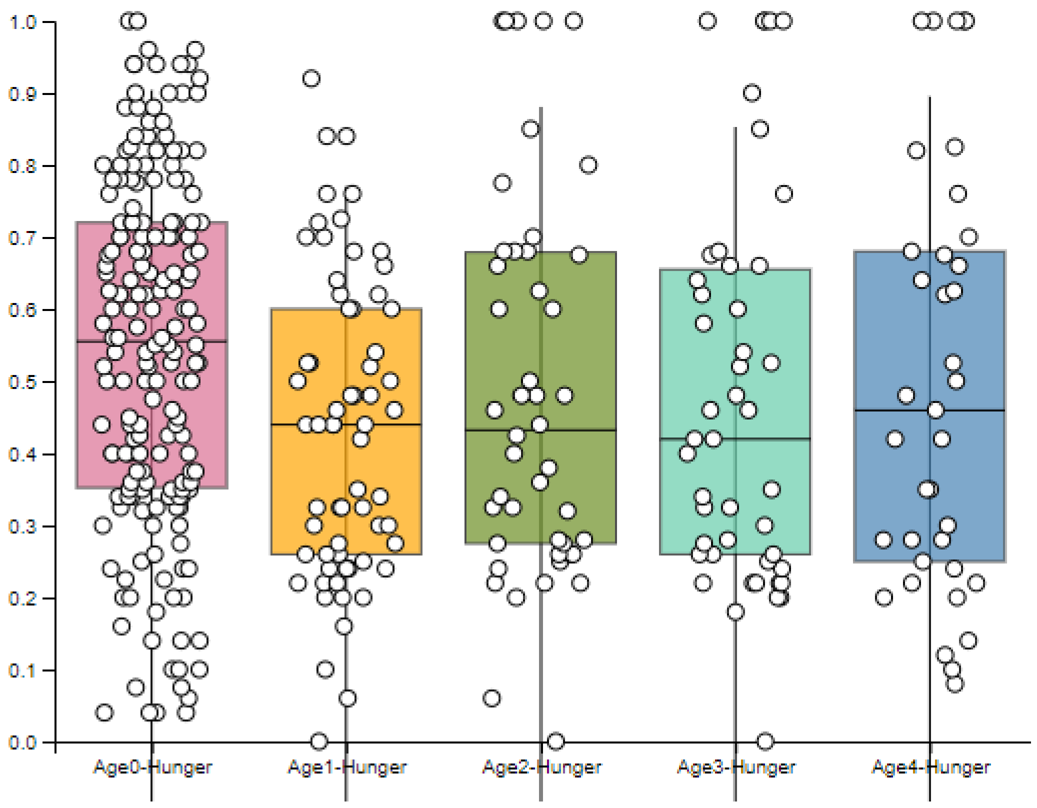


Figure 4.1: *Box-and-whisker plot showing general distribution of hunger values at the end of each age. Notice how the distribution remains roughly the same size throughout all of the ages (D3).*

The box-and-whiskers plot for the population statistics (**Figure 4.2**) show quite a different scenario than with hunger. The main takeaway from this plot is that the range of population values grows with each sequential age. Although the average recorded population drops slightly after the first age, it then slightly increases with each age, with the biggest increase being between the second and third ages. Similar to the hunger distribution, the amount of outliers with the population statistic also increases, and about half of the collected population values are outliers by the final age.

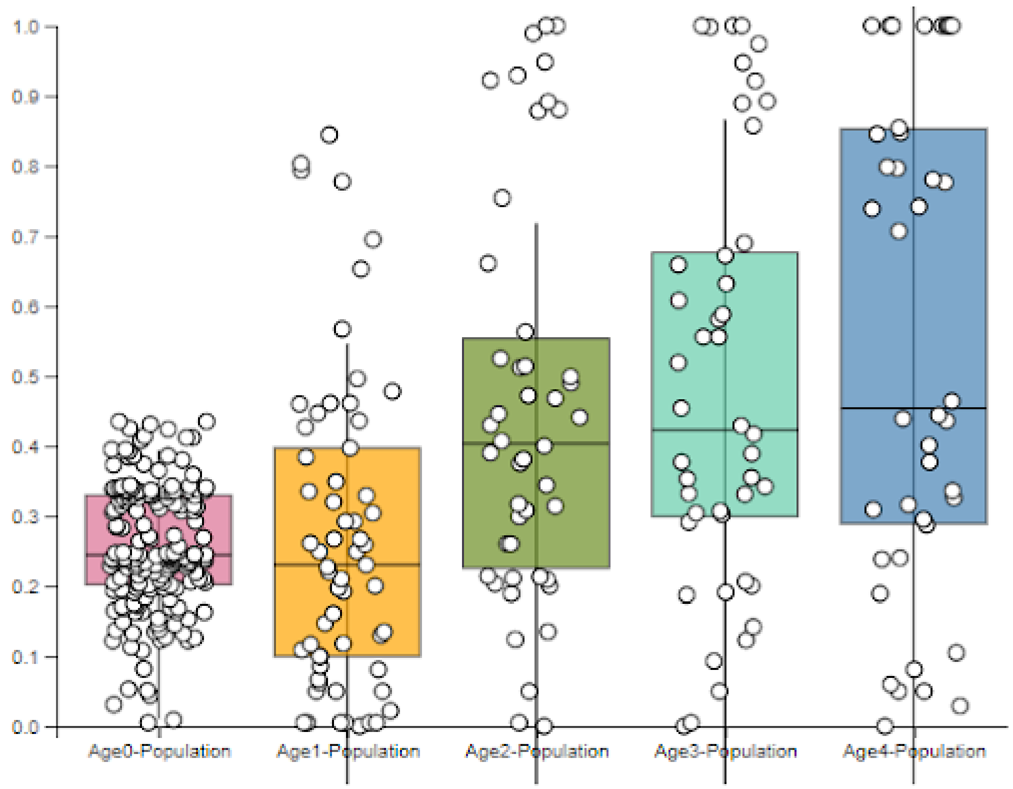


Figure 4.2: *Box-and-whisker plot showing general distribution of population values at the end of each age. It is interesting to note that the distribution grows in size with each passing age (D3).*

The distribution for the military statistics, as detailed in the box-and-whiskers plot in **Figure 4.3**, is not consistent at all. Unlike with the population statistical distribution where many of the recorded values are clustered together, the military values by the end of the first age are much more spread out. Both the range and average recorded military statistics take a huge dive down by the end of the second age and remain consistently low for the next couple of ages. By the end of the final age, the average military value is extremely low, but the range is so great due to the number of large outliers either at 0% or at 100%. Out of all of the statistics with box-and-whisker plots, military has the greatest range at the end of the final age.

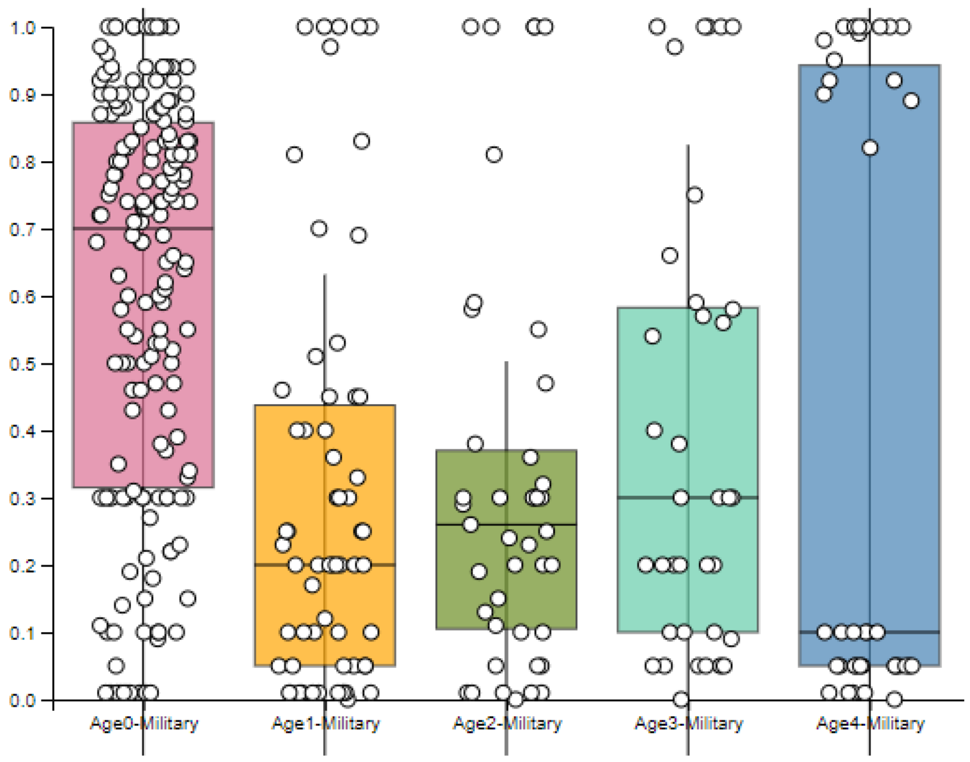


Figure 4.3: *Box-and-whisker plot showing general distribution of military values at the end of each age. The number of outliers for the military values by the end of the final age is especially interesting (D3).*

The science statistics generate a unique box-and-whiskers plot, which is visualized in **Figure 4.4** below. This plot may be the most interesting out of them all: the range of science values decreases with each age as almost all of them converge to 100%, and by the final age, the range is very small. The average science value increases by roughly 0.3 over the first few ages and reaches a maximum by the end of the third age. Although there are a few outliers in each age after the first one that have relatively lower science, the plot shows that most playthroughs of the game ended up with almost full science.

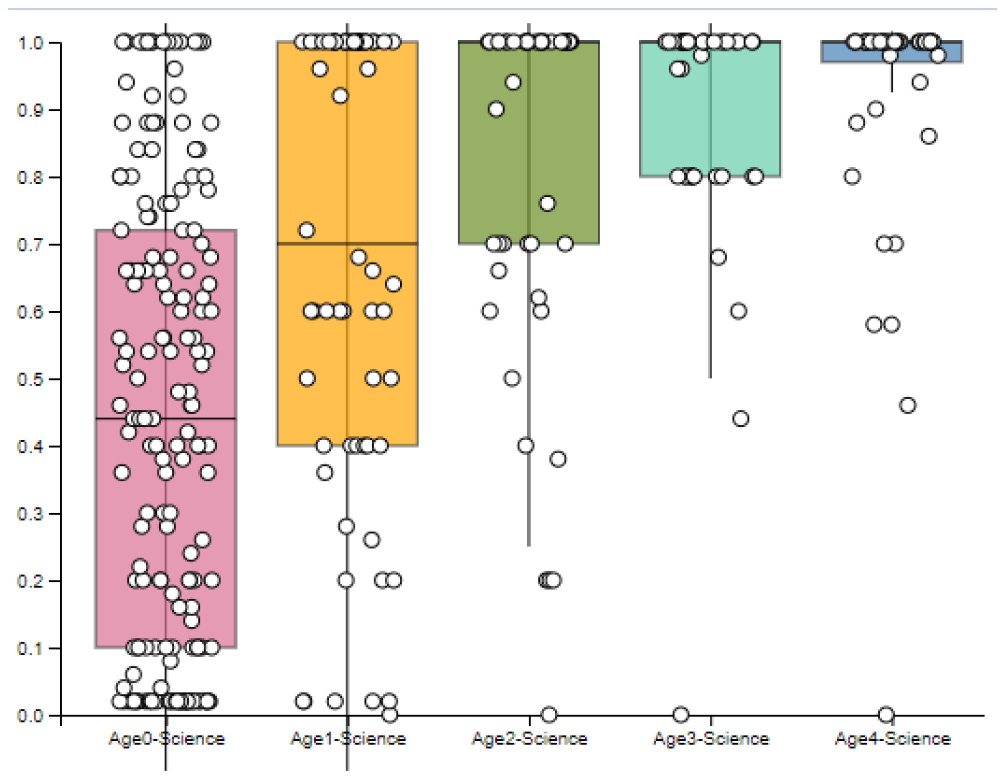


Figure 4.4: *Box-and-whisker plot showing general distribution of science values at the end of each age. Notice how the recorded science values slowly max out with each successive age (D3).*

The last statistic we gathered data on is the player's security levels throughout the game. As seen in **Figure 4.5 below**, the box-and-whiskers plots are difficult to read. This is primarily due to the fact box and whisker plots are used to visualize continuous data, rather than discrete data. Since the security statistic can only be an integer from 1 to 5, the data is discrete and therefore cannot be described well by a box-and-whiskers plot. A stacked or grouped bar chart would likely show the data in a much more informative way. However, you can still observe in the plot below that players for the most part kept their security high when they had the chance to upgrade, and by the end of the final age most players had a security level in the upper range (3 to 5).

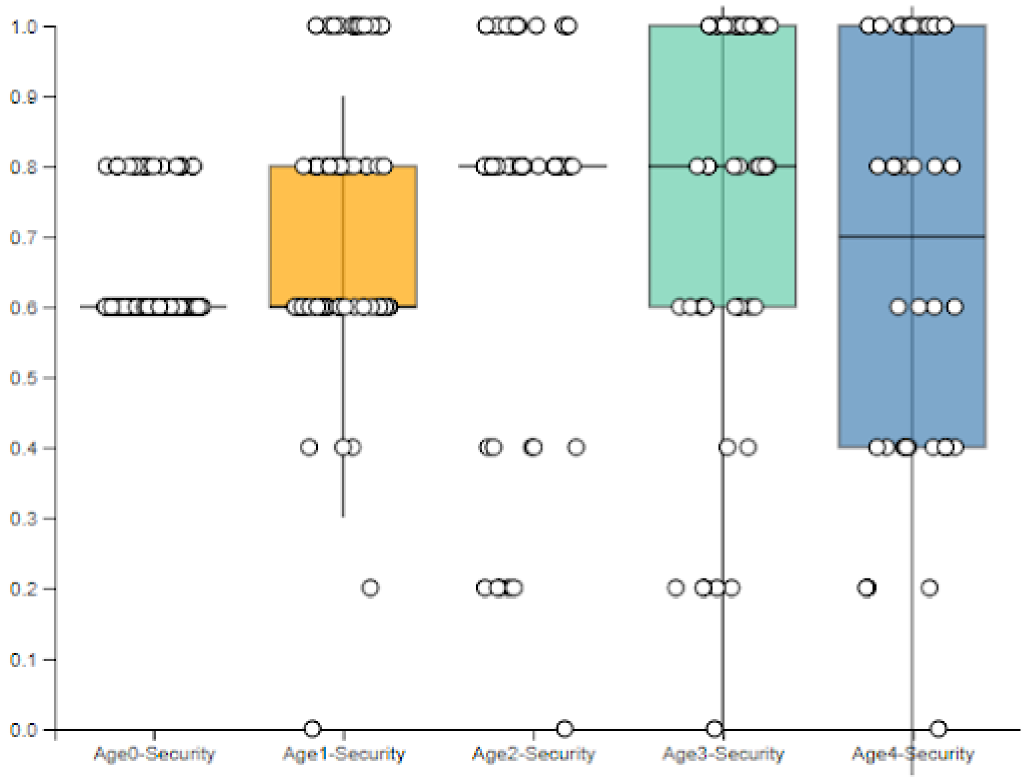


Figure 4.5: *Box-and-whisker plot showing general distribution of security values at the end of each age. Although not the best plot for the discrete security values, it is still possible to see that security was kept on the higher end by most players (D3).*

Now that it is possible to see visually how players performed in the game, the next question to answer reflects the main goal of our project: did the visuals help the players make informed decisions? Although the above box-and-whisker plots certainly provide some insight into the effectiveness of the visuals, additional graphs and quotes from the survey will provide more of the players' perspectives and better answer the question that dictated our project.

It is important to note that previous familiarity with visuals in a programming or gaming context may introduce a slight bias into the results, so we asked players in the survey if they had any experience with these types of visuals. The pie chart in **Figure 4.6** below shows the responses. Slightly more than half of the players were familiar with visual feedback similarly to how it appears in our game, and another 39% said they knew about some basic visuals, leaving only 5% of players who had zero prior experience.

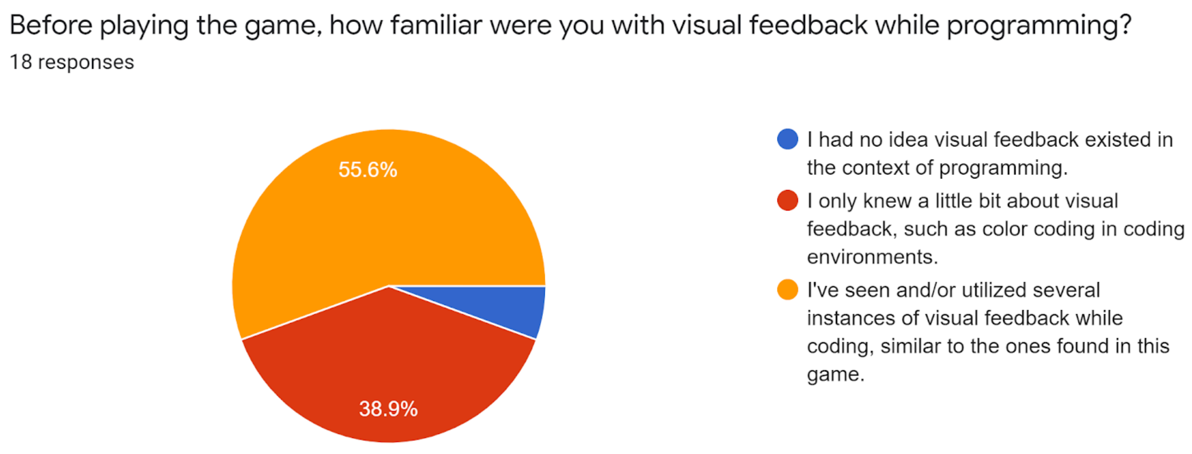


Figure 4.6: *Pie chart detailing the amount of familiarity players had with visuals in programming. More than 90% of the players had at least some experience.*

Although prior knowledge of visual feedback may influence the results quite a bit, they will have little impact if the visuals themselves were not understandable. Between the bar graphs representing the amount of each statistic, the timer, the color coding, the legend, and the sparklines showing the change in each statistic over the last few prompts, all of the visuals within the game were meant to influence the decisions that players made. Some of the quotes pulled from the surveys communicate whether the visuals helped each player make decisions or not:

- "I didn't remember what all of the health/food/other bars stood for."
- "The visuals definitely helped in understanding the current state of the game."
- "Some of the bars were unclear as to what they were."
- "The different prompt colors were nice, at least to differentiate the text."
- "It took a little time to know what the icons meant on the health bar, but it was great once I understood it."

The quotes are quite varied, with many players clearly seeing the correlation between the visuals and their decisions in the game, while others were more confused about the purpose of the visuals. The bar chart in **Figure 4.7** below shows a similar result, where the players were asked on a scale from 1 to 5 of how easy it was to understand how the results of their decisions impacted the rest of the game. Exactly half of the players were able to see the impact their decisions had on gameplay, while the other half either were in the middle or had no idea what their decisions actually did in the game.

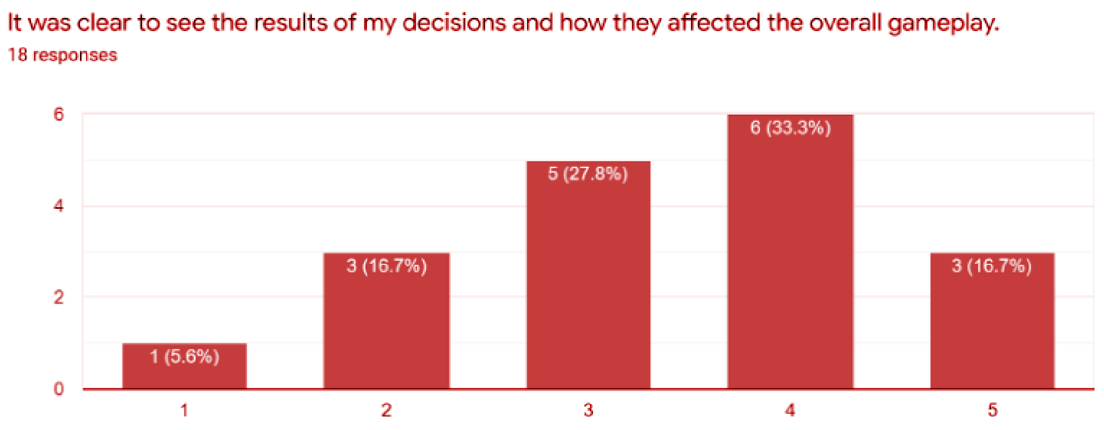


Figure 4.7: *Bar graph of player's feedback on whether they could understand their decisions and resulting consequences clearly. Only 22.3% of the players had trouble understanding the results,and half of them seemed to understand their impact clearly.*

Although many players seemed to be using the visuals to view the results of their decisions, this same level of understanding did not apply to other elements of the game. One major option that players had in the game was to use past decisions, either by looking at the trends of each statistics with the sparkline visuals (the easier way) or scrolling back up to previously-answered prompts (the more difficult way). As the bar chart in **Figure 4.8** below shows, not many players appeared to look into the past for making a current decision. In fact, more than 50% of players did not, and only approximately 17% of players did.

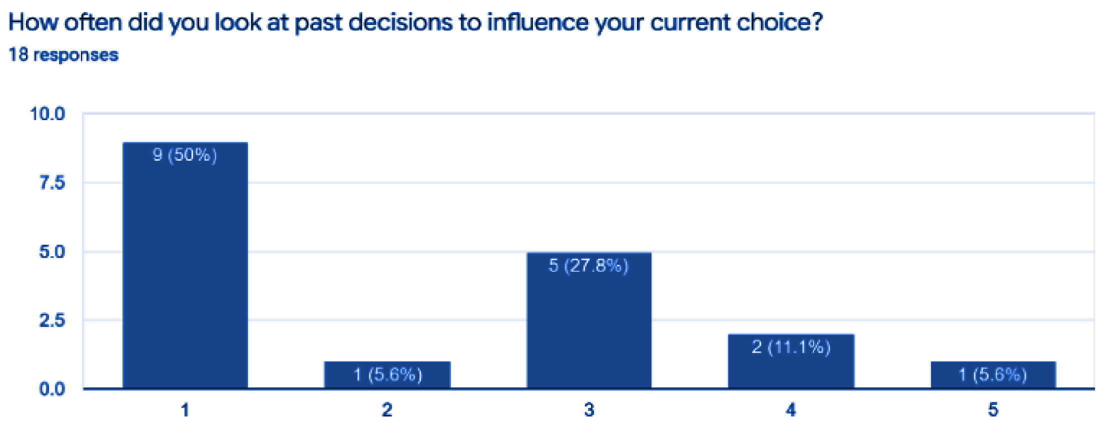


Figure 4.8: *Bar chart showing whether players used past decisions to influence their current choice. More than 50% of the players never looked into the past.*

The visuals within *Coding Through the Ages* are intended to show players how all of the statistics change with each decision, and the above results provide a good idea of whether the visuals are effective in that regard or not. However, the visuals within the game are not static; rather, they change with time, which is a major component of the game. Therefore, there is one more crucial question to ask with regards to the visuals: did they help players manage their time?

Every person has differing management skills, especially in regards to stressful tasks. It is difficult to provide a generalized conclusion for a whole group of people regarding time management, so we pulled together a pie chart shown in **Figure 4.9** below where we asked how fast of a reader each player is. Although reading the prompts is not the only component of the game that requires quick thinking and acting, it is certainly one of the aspects that could take a lot of time. Almost half of our players could interpret text after reading fairly quickly, and interestingly one third of the players were very slow readers. This last observation stressed the need for more visual annotations to further help manage time in the game.

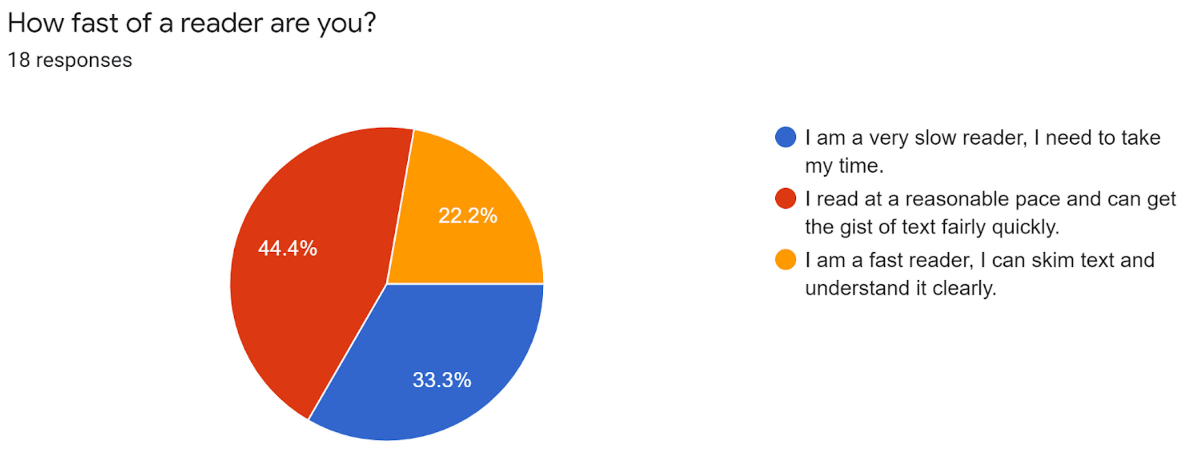


Figure 4.9: *Pie chart of player's self evaluation on their reading speed. One-third of the players claimed to have slow reading skills, while the rest claimed to have average-to-above reading speed.*

If there were no visuals in the game and players had to fully read each prompt in order to make an informed decision, we could expect to see extremely varied results: players who could read faster more than likely be less stressed out about the time aspect of the game than players who needed to take their time to read. However, the visuals were placed into the game with the intention of allowing the players to better manage every component in the game and not be as stressed with finishing everything in time. The following quotes taken from some survey responses reveal some of the players' opinions on managing time in the game:

- "I didn't feel like I had enough time to code the functions."
- "Having it that fast was a bit challenging, but that also added an appeal because you had to stay on the ball."
- "The game was so fast that I do not remember any of my decisions."
- "It seemed that everything moved in a good pace and in the right direction."

- "I was too focused on the health bar. It seemed to fall too quickly."

Once again, the responses from all of the players of our game were completely different, with some people feeling comfortable (likely not at first, but shortly after getting used to the mechanics) with the amount of time they had while many others were stressed out at how little time they had with so many pieces to manage. This sentiment is reflected in **Figure 4.10** below, which is a bar chart showing on a scale of 1 to 5 if players felt they had enough time for completing the different tasks in the game. The results were pretty evenly split: about one third of players did not feel like they had enough time, another third of players felt that they had plenty of time, and the rest were in the middle. It is important to note how, especially as the above quotes explain, some components of the game, such as automating functions and reading the prompts, required more time than other tasks. The chart below summarizes the time over all tasks in the game, rather than individual ones.

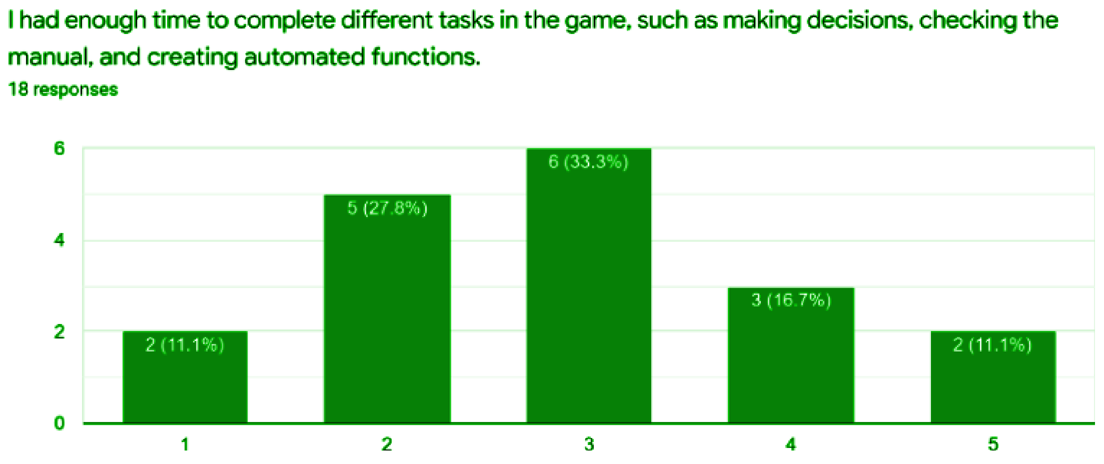


Figure 4.10: *Bar chart showing player feedback on whether they had enough time to complete different tasks during the course of the game The results were pretty evenly split, with some players being more comfortable with the amount of time than others.*

To further analyze the amount of time that players needed to progress in the game, we plotted an event graph in D3 of the final run of several playtesters, which is shown in **Figure 4.11** below. (Unfortunately, due to data collection issues, the last decision of each age was lost.) Each row represents the timeline of a player's run-through of the game, with each dot presenting when a decision was made. Each age is a different color, and the first dot in each age is bigger to represent when the first decision in each age was made. It is difficult to read the graph as is since it is quite long and plotted over the entire course of gameplay, so we will break it into chunks and analyze the most important observations. The entire event graph can be found here.

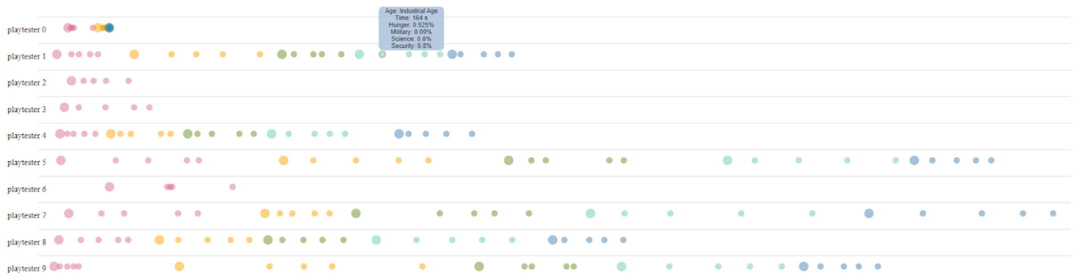


Figure 4.11: *Event graph showcasing when decisions were made by different playtesters on their last playthrough of the game (D3). To view the graph in a bigger window with full interactivity, click here.*

Figure 4.12 below shows a zoom-in of the functionality of the event graph when navigating to the link provided above. When hovering any point (decision) in any playtester's timeline, a small box listing what age the player was in, at what time the decision was made (in seconds), and the amount of almost every statistic they had at the time is detailed. This makes it possible to observe specific details about player choices at any point in time.

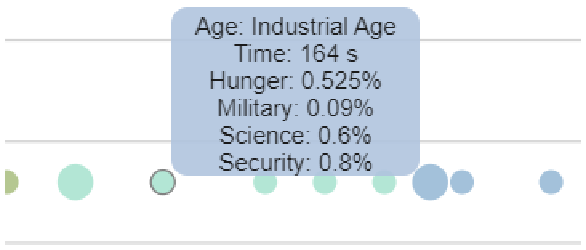


Figure 4.12: *Zoom in of the event graph hovering over a specific decision, revealing the age, time, and some of the statistic values recorded (D3).*

There are a few key observations that can be drawn from the event graph in terms of time management in the game. One obvious thing to note is that some players made decisions more quickly than others, which confirms the variety of responses we got from players before with the graphs and quotes about having enough time. However, we noticed this difference especially after the start of the second age. To highlight an example, **Figure 4.13** shows just how much more time playtester 9 took to make some of their decisions than playtester 8 did by comparing the first few ages in their gameplay. Playtester 9 had large gaps of time in between finishing the first age and deciding on the second age, and they hadn't even finished the second age by the time playtester 8 finished the third age.

Another interesting thing to note is that some of the playtesters shown here did not complete the game on their last run through. Specifically, a few of them lost the game at the end of the first age. As clearly shown in **Figure 4.14** below, playtesters 2 and 3 ran out of hunger curiously right in between the first and second ages during their final experience with the game. Playtester 6 also lost the game at that same spot, although it is not shown in the figure. However, notice the exact times at which those three players made decisions are not the same, so the amount of time they had to make the decisions may not have been as much of a factor as one would initially believe.

One final observation to make is that on their last gameplay session, a player (playtester 0) who was skillful in using JavaScript decided to experiment with the automation function. This playtester wanted to explore the automation to its fullest and used it to run the choose() command automatically. We did not expect players to use the automation functionality for this, and were very surprised to find that the method essentially broke the game. Playtester 0 was able to zoom through all 20 prompts and finish the game in less than 30 seconds, as seen in **Figure 4.15** below.

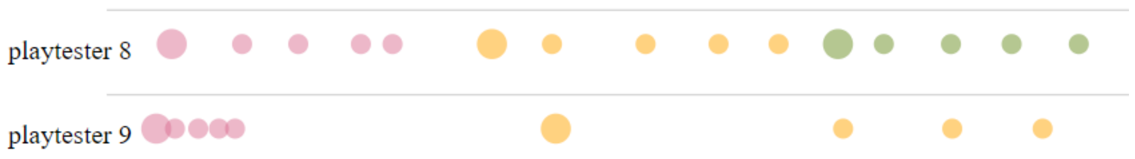


Figure 4.13: *Comparison from the event graph between playtester 8 and playtester 9's time spent making decisions. Pay particular attention to the large gaps in time found in playtester 9's data (D3).*

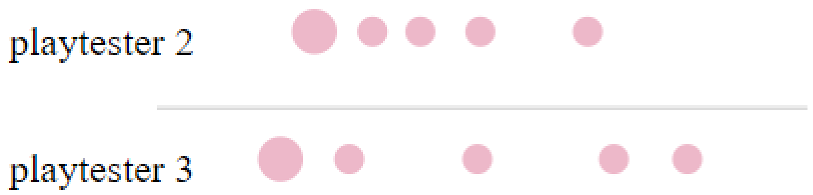


Figure 4.14: *Close up of the event graph showing the last runs of both playtesters 2 and 3. Interestingly, they both received a game over after finishing the first age (D3).*

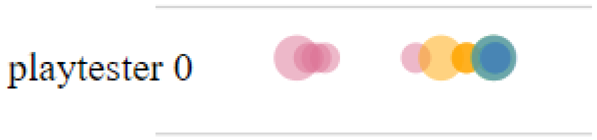


Figure 4.15: *Playtester 0, a player who went out of curiosity and automated the choose() function. The game ended in less than 30 seconds, and all statistics stacked upon each other (D3).*

Finally, based on all the data we gathered from players in terms of their performance, their understanding of the visual feedback, and their ability to manage time, we could gauge their overall enjoyment of the game and its mechanics. We asked players directly in the surveys how they would describe their gameplay experiences, and these are some of the responses we received:

- "I had a good experience overall. The visuals were good. I liked the simplicity in terms of keeping the game to a down-scrolling text game. The buckets/bins showing the levels for each stat could've been bigger though. The icons for each stat are also hard to see in the persistent status bar."
- "The gameplay was okay. I personally think that the timer is way too short for someone to actually fully experience the game. For me, I skipped reading a lot in order to make my next choice by the time the timer ran out. When I tried to do automated functions I ran out of time. I did much better in the game by not using automated things."
- "It felt like a programming environment. I liked the dark theme. Sometimes I wished the text was smaller or there was less text. I also think additional space between the output and prompt are needed. eat() & secure() should be required to type in each time and be cleared from the prompt."
- "I thought it was a really cool concept. I wished there were some more graphics, but because it's a coding based adventure game, I understand why there's less visuals."
- "The game was fun, the options were simple but diverse enough to think about each choice a bit. The visuals helped understand the health but my main focus was on the timing bar at the bottom of the screen."

Each player had their own level of enjoyment from the game, similarly to how varied the rest of our results were. Some players were able to manage time better than others, some were more informed of the impact their decisions had on the remaining time in the game, and some players experimented with different techniques that we did not even think of ourselves. Since there were so many tools at the player's disposal within the game, they each had their own unique experience and therefore enjoyed the game to a different degree than others. Based on the information presented here, we can make several predictions and theorize why the gameplay experiences for each player ended up the way they did.

Chapter 5

Discussion and Conclusions

5.1 Discussion

Based on the gameplay results gathered from all of our playtesters, it is clear to see that each player had a different experience within the game. Formulating all of the graphs from the collected data revealed just how mixed the results were. In almost every category, player opinions were either quite positive or fairly negative, and the quantitative data was evenly split between higher and lower values. Therefore, as an overall statement, we can assume that our implementation of the game was not the most effective, but that is not to say the visuals did not assist the players in making informed decisions.

Many players were able to progress through the game, although most gameplay sessions ended in a game over. With each successive age, less and less decisions were recorded since more and more players were losing the game and not progressing. During our playtesting sessions, we observed that many first-time playthroughs of the game resulted in a game over within the first few prompts. This seemed to be mostly because the players had a hard time understanding the mechanics of the game within the time limit at first. However, when players restarted the game, in general they got closer to the end, and most players completed the game before the end of each session. Although it was not our original intention when designing the game, *Coding Through the Ages* ended up requiring several tries to master, and this may suggest that more game balancing was needed.

Going into further detail about each statistic, the hunger statistic distribution remaining relatively the same shape across each age implies that players were able to manage it quite consistently. Once players understood that the hunger bar reaching 0 meant game over, it seems that they kept an eye especially on that hunger bar in the visual and did not let it get too low. As we observed during our playtesting sessions, some players would eat whenever they had food and their hunger was not at the maximum, and others would play a riskier game and let the hunger get really low before spamming the eat() command. The red coloring, positioning, and relatively fast decrease of the hunger bar may have allowed players to pay attention to the current status of their hunger, as **Figure 5.1** below illustrates.

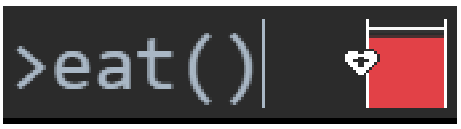


Figure 5.1: *The hunger bar was intentionally designed to keep players' attention the most out of any statistics, as it was the first bar in the visual and also colored a deep red.*

There was much less consistency with the population, military, and science statistics, especially since those values were affected by the decisions chosen by the player. It is difficult to say based on the box-and-whisker plots whether or not the players used the bar visuals to intentionally choose certain stats over others. On average, however, science was quite high for most players while both military and population were varied depending on the decisions made. This could mean either players prioritized science or the decisions in the game favored science too much and were not fully balanced. There were a few possible decisions throughout the game that increased science by a large amount, so it is definitely possible that each player just happened to choose those options without any true intention of increasing science. **Figure 5.2** shows an example of change indicators showing how science could drastically increase while population and military did not change by much.

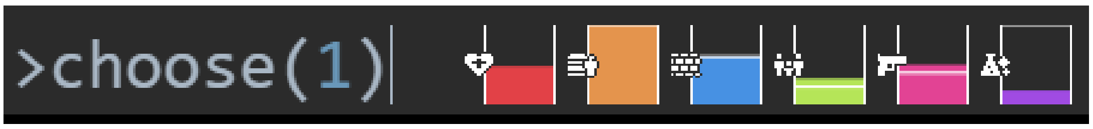
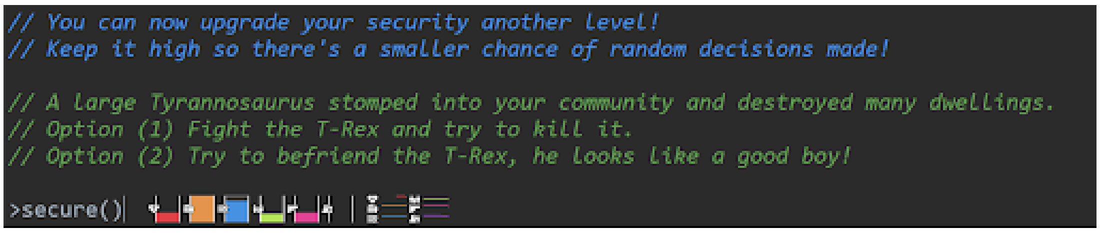


Figure 5.2: *An example of an option where science would increase by a lot by military and population would not, which could have biased recorded science values due to less balancing in the game.*

We can make a fairly accurate statement about the distribution of security values though. Almost consistently for every player over the course of the game, security was kept in the upper range from 3 - 5. This likely means that the prompts alerting the player to increase security when they were allowed to did indeed work, especially with the blue color coding to differentiate it from the rest of the in-game text. We cannot definitively claim that players understood what keeping security high actually did within the game (especially because several players claimed they didn't understand it), but they were able to recognize that it was an aspect to keep a close eye on. **Figure 5.3** shows an example of how the blue security text would cause a player to want to increase security right away with the secure() command.



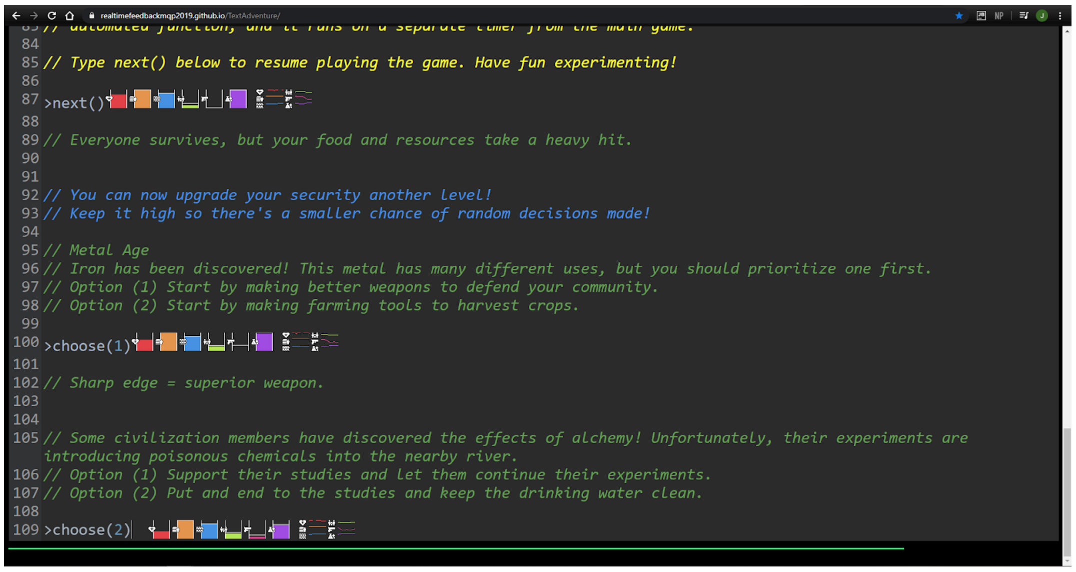
```
// You can now upgrade your security another level!
// Keep it high so there's a smaller chance of random decisions made!

// A large Tyrannosaurus stomped into your community and destroyed many dwellings.
// Option (1) Fight the T-Rex and try to kill it.
// Option (2) Try to befriend the T-Rex, he looks like a good boy!

>secure()
```

Figure 5.3: *An in-game scenario that we noticed many times during playtesting, where a user would instinctively use the secure() command right after seeing the blue text appear.*

So the question still remains: did the visuals help the player make informed decisions in the game? The short answer is that they did in some areas and not in others. The fact that every player was a computer science major and that most were already familiar with some amount of visual feedback obviously helped them to ascertain the purpose of the in-game visuals. The most prominent visuals in the game were the bars corresponding to each statistic, so we hoped that those visuals especially would influence each decision the players made. According to our gathered results, most players could see how their choices affected the gameplay, which means that they definitely noticed the inline bar visuals change with each decision made.

However, a few players mentioned in the survey responses that they were not exactly sure what some bars were supposed to represent, and other players reported that they were choosing between options almost randomly without even reading the prompts. Clearly the connectivity between each bar visual and the options in each prompt was not made clear enough during gameplay, and this affected some players more than others. Despite this, on the whole players relied more on the bar visuals for understanding the current context of the game rather than reading all of the text in the prompts, which in turn sped up their gameplay, so they did indeed influence decisions made, just not to the exact degree we had hoped for.

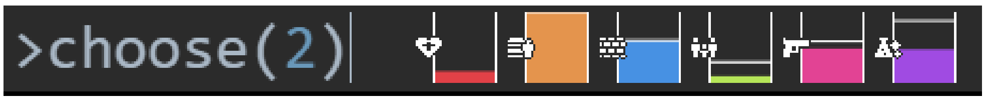
Unfortunately, not all of the visuals in the game were utilized by the players to the extent of the bar visuals. The sparklines especially went almost completely unnoticed. They captured the change in each statistic over the last 5 decisions made, so they were intended to give players some guidance in making a choice by showing the previous history of each statistic. However, almost no players used prior statistic values to make their current decision, and we believe this may be because the sparklines were too small and not easily distinguishable to the player. This is further proven by the fact that no players even mentioned the sparkline visuals in the survey responses, so they may not have even noticed them. The sparklines were finalized fairly late into development and only a couple of weeks before playtesting, so with further refinements they may have been a more prominent feature within the game. **Figure 5.4** below displays an entire screenshot of the game with the bar visuals and sparklines and highlights how small the sparklines really are.



```
// automated function, and it runs on a separate timer from the main game.

// Type next() below to resume playing the game. Have fun experimenting!

>next()

// Everyone survives, but your food and resources take a heavy hit.


// You can now upgrade your security another level!
// Keep it high so there's a smaller chance of random decisions made!

// Metal Age
// Iron has been discovered! This metal has many different uses, but you should prioritize one first.
// Option (1) Start by making better weapons to defend your community.
// Option (2) Start by making farming tools to harvest crops.

>choose(1)

// Sharp edge = superior weapon.


// Some civilization members have discovered the effects of alchemy! Unfortunately, their experiments are
introducing poisonous chemicals into the nearby river.
// Option (1) Support their studies and let them continue their experiments.
// Option (2) Put and end to the studies and keep the drinking water clean.

>choose(2)
```

Figure 5.4: *As is clearly (or rather not so clearly) illustrated, the sparklines are quite small compared to other components in the game, which may have led players to overlook them.*

Two other visuals that were seemingly rarely used, but more so than the sparklines, were the change indicators on each bar (the white boxes for each bar before making a decision showing how choosing that option will increase or decrease the bar), and the legend. There were only a few instances where we noticed a player looking through both options for a prompt and seeing how each bar would change before making a final decision. Most players made a decision and executed the choose() command without considering both options, possibly because they did not notice the change indicators. The indicators were small white outlines, and especially since the bars themselves were limited in terms of width and height, they may have gone unnoticed. A flashing animation above the bars may have been a better implementation of signifying a change. **Figure 5.5** shows the potential of the change indicators to show the player how options will impact statistics, but also demonstrates how faint the lines can be.



```
>choose(2)
```

Figure 5.5: *An in-game example of the change indicators appearing when typing the choose() command with an option specified. Notice how the white indicator lines somewhat blend with the bars and can be quite difficult to discern.*

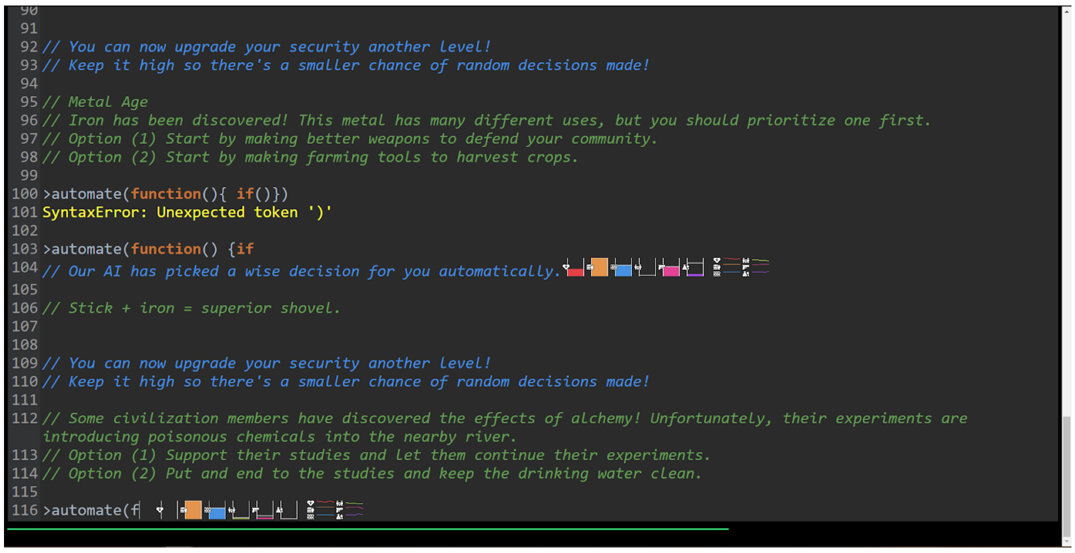
Also, not every player remembered the legend() command, which may have been because players did not fully take the time to understand what each bar meant, or because the legend() command was the last command mentioned in the manual (before automation was unlocked).

The players who did use the legend found it to be very useful, especially with the pixelated icons matching and color coding matching the bar visuals.

The color coding and the timer were the two visuals that players understood best. They also happened to be the simplest, which definitely contributed to their successful interpretation by players. The unique color for each bar in the bar visual was apparent enough so that players were able to tell that each bar represented something entirely different. The coloring with the text especially helped to communicate to the player if a special event within the game was happening, such as with the blue text indicating security could be used or the yellow text explaining the amount of automated functions a player could write. As mentioned before, the colored text within the legend further surfaced the meaning of the bar visuals. In addition, the timer was simple and always present at the bottom of the screen, so it clearly conveyed to players the sense of urgency that made up the foundation of the game.

The amount of time players had was made apparent by the simplistic timer visual, but did players feel like they had enough time to complete each task within the game? Once again, the answer is not as simple as yes or no, as the results were fairly evenly split. Some players felt they had enough time to do different tasks, especially making decisions, and those are more than likely the players who claimed they were faster readers in our surveys. Others were very stressed out by the time limit, as several of them mentioned in the survey responses, and they may have very well been overwhelmed by the presence of the time limit within the game. Stress related to limited time was definitely an aspect we wanted within our game, but the idea was that the visuals would help mitigate this stress by streamlining the decision process and making it much easier to see the effects of decision-making.

Some players took much longer to make decisions than others. As pointed out in the event graph in the Results section, several players spent a considerable amount of time between decisions after the end of the first age, and we believe the reason for this might be related to automation. The automation functionality is introduced right after the first age, but when the tutorial is displayed, the game timer pauses. Right after several players passed through the automation tutorial and began the second age, they most likely tried experimenting with the automation before making a decision. Other players did not do this however, and as we noticed in the survey responses, many players felt that the automation was not explained clearly or that they did not have enough time to stop and try messing around with it. Some players even received a game over right after the first age, possibly implying that they tried experimenting with automation and forgot about keeping their hunger up. **Figure 5.6** below shows an example scenario where tinkering with automation too much could result in a game over.



```

// You can now upgrade your security another level!
// Keep it high so there's a smaller chance of random decisions made!

// Metal Age
// Iron has been discovered! This metal has many different uses, but you should prioritize one first.
// Option (1) Start by making better weapons to defend your community.
// Option (2) Start by making farming tools to harvest crops.

>automate(function(){ if()})
SyntaxError: Unexpected token ')'

>automate(function() {if
// Our AI has picked a wise decision for you automatically.

// Stick + iron = superior shovel.


// You can now upgrade your security another level!
// Keep it high so there's a smaller chance of random decisions made!

// Some civilization members have discovered the effects of alchemy! Unfortunately, their experiments are
introducing poisonous chemicals into the nearby river.
// Option (1) Support their studies and let them continue their experiments.
// Option (2) Put and end to the studies and keep the drinking water clean.

>automate(f
```

Figure 5.6: *An example scenario within the game where a player could be getting used to the automate() function and experimenting with its capabilities and not paying attention to their hunger.*

We believe the examples from the event graph where players took more time between some decisions represent the players who experimented more with automated functions. We can almost guarantee this is the case for the first playtester in the event-timeline graph, where they completed the game in record time, and many of the recorded decision points appeared to overlap within tiny increments of time. We had one of our players actually automate the choose command (executing choose(1) whenever they could), and we believe that run is shown as the first entry in the event graph. The automation was a bit confusing for some people, as it could've been explained better, but for those who did use it we can tell that overall their gameplay session was longer due to less time-related stress.

Many players were frustrated with what little time they had to complete each task, and many especially felt they did not have enough time to try automation or to read each individual prompt. However, that did not mean they did not enjoy the game. We got many positive responses from players explaining that the game was a really neat concept and that in general they understood how the visuals worked within the game. Some people even mentioned wanting more visuals to refine the gameplay process even more. Even though we developed a unique and fun computer game with semi-effective visual annotations in Coding Through the Ages, there are plenty of tweaks and additional elements that could be incorporated to create a more enjoyable player experience.

5.2 Recommendations

As we developed *Coding Through the Ages*, we had many creative ideas that we unfortunately did not have the time or resources to implement within the game. Some of these features include the following:

- Branching story paths
- Official experimental setup
- Prompt parsing
- More automation functionality

Branching story paths was something we planned on including from the very beginning. We envisioned a player following a different sequence of prompts depending on the decision they chose, so the changes in statistics would not be the only difference between options when responding to a prompt. Eventually, the story paths could converge to prevent too many different endings from needing to be within the game. This would be extremely easy to implement, as the prompts for the game are stored in a JSON object, and each prompt has a field for which prompt to go to next. Any future developer could simply add a new prompt into the correct age and specify the name of that new prompt as the destination. **Figure 5.7** below shows a very simple diagram with an idea for diverging and converging story branches.

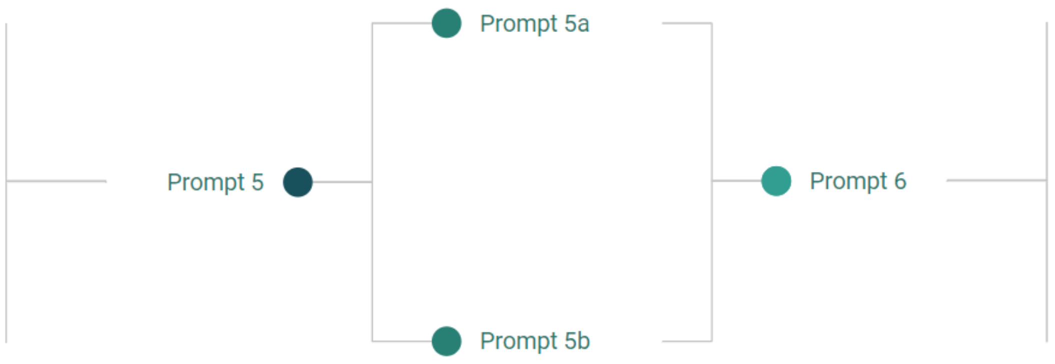


Figure 5.7: *Diagram showing an example of branching story paths based on the decisions made by a player.*

Originally, we wanted to set up an actual experiment for our game, where different versions of the game had varying amounts of visual information presented to the player. One version would have no visuals and only rely on reading the prompts, another version had a limited amount of visuals where the user could be told which stats would change from there decisions, but not by how much, and one final version with full visuals that would have been very similar to our current game. We planned on doing in-person testing as well as crowdsourcing (the crowdsourcing research can be found in **Appendix D**). However, due to time constraints, the drastic gameplay effects with a lack of visuals, and COVID-19, we decided to focus on just creating an enjoyable game with a maximum amount of visual annotations and hold playtesting sessions remotely. This is something that could definitely push our game and visual feedback research forward, as it would be possible to directly show how players performed with visuals as compared to without. An extremely simple diagram detailing different experimental groups for different versions of our game is shown in **Figure 5.8** below.

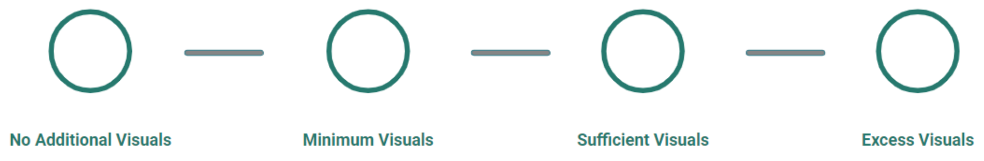
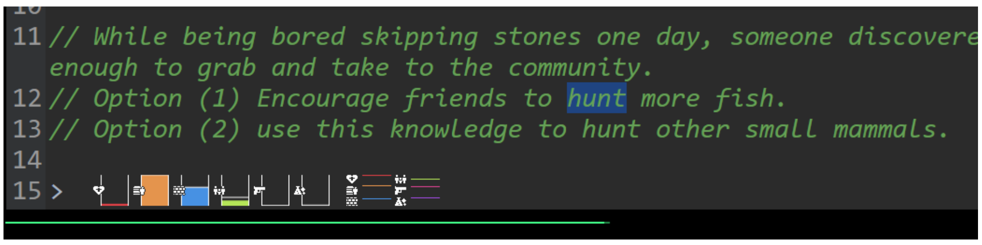


Figure 5.8: *Very simple diagram of ways to make different versions of our game in an experimental procedure.*

Although one of the main intentions of the bar visuals was to decrease the amount of time players needed to spend reading the prompts, we had originally wanted the prompts to play more of a role in the game. One idea we had that would've been interesting was the ability to parse the prompts - essentially pick out keywords from the prompt descriptions or options and make a decision based on the words picked. For example, as shown in **Figure 5.9** below, the word "hunt" is mentioned, which implies that food will be gained for whatever option that word occurs in. In this case, both options have it, but players would parse the prompt with an automated function, check to see which option has the word "hunt" or another food-related word, and make the decision based on that. Other keywords would be available as well if players wanted to prioritize decisions for other statistics.



```
// While being bored skipping stones one day, someone discovere
enough to grab and take to the community.
// Option (1) Encourage friends to hunt more fish.
// Option (2) use this knowledge to hunt other small mammals.

>
```

Figure 5.9: *Example of possibly including parsing prompts to make decisions. In this case, the player could look for a word such as "hunt" to make decisions based on increasing food.*

While on the topic of automation, a majority of the players used the automation function purely for calling the command eat() repetitively so they would not have to worry about getting their hunger too low. This also happened to be the example provided in the automation tutorial, and many players copied and pasted the example for their own use. Many of them commented in the survey responses that they could not think of any other uses for automation besides using the eat() command when hunger was low enough. Extending this functionality to other areas of the game besides the built-in commands and letting users write their own functions or create their own variables would make the automation be an even more user-friendly and experimental tool to utilize to the player's advantage.

While we had several of our own ideas on how to add the *Coding Through the Ages*, our playtesters also had a few ideas on things we could add or improve upon. Some of these ideas include the following:

- "Maybe the legend could be on the side with the labels. Maybe that will make legend() command useless but it would be more user-friendly."
- "Since I am fairly familiar with using command line, I often expected the instruction I wrote to disappear, and to be able to make it reappear by using the up arrow key. Tab complete would be nice also."
- "I think a tutorial round where you actually type out each of the functions would be nice. I think the green and yellow [text colors] were fine. It would be nice to have maybe an API type thing at the top of the screen."

If the player needs to understand in text what each icon and color in the bar visuals actually means, they need to call the legend() function. This presents some usability issues, as a few players noted, since it requires the player to type another command to display. Originally, we had thought of having a "cheat table" with brief explanations of the commands and statistics on the side of the interface, which would also contain the legend. This would allow players to look up the meanings of each statistics and command without being interrupted from flow of the rest of the game. Other improvements to the visuals, such as increasing the height for the bar visuals and using more color coding with aspects like the change indicator, could augment the game even further.

The other major change players suggested was making the game more like an actual command line. The eat() and security() commands did not disappear after typing them and hitting enter, like most commands in a terminal would. The choose() command did disappear, but the reason eat() and security() didn't was so players could repeatedly execute them without having to retype them. Having other pieces of functionality like hitting the up arrow to reuse a previously-typed command and having a "cheat sheet" on screen to access as a mini API could further immerse the player in the game experience and make them feel as though they were interacting with an old computer terminal.

Many additional components have the potential to make *Coding Through the Ages* an even more special experience. Some can be incorporated to improve the clarity in visuals, some can further envelop the player in the game's narrative, and some can be general bug fixes that would make the gameplay experience slightly less cumbersome. Despite the benefits these additions would serve, the game we ended up creating seemed to meet our overall goals.

5.3 Summary

With this project, we set out to develop a coding-based game with visual annotations that would help the player make informed decisions throughout the course of the game. We feel that we accomplished this goal with all of our visuals, especially the bar visuals, timer, and colored text. Although many players did not even notice the sparklines, change indicators, and legend() command visuals, all of the data we gathered and visualized along with survey responses proved that the combination of all the visuals helped players with both time and resource management to some degree.

There is still so much to explore within the area of visual feedback in games, and it's surprising that not many examples besides the ones mentioned in our background research don't utilize the true capabilities of these types of visuals. There is so much potential for coding-based games with inline visual annotations, and the different possible applications for that kind of research seems endless. With careful planning, multiple iterations of design and implementation, and observant testing, we were able to scratch the surface of what could be a very unique genre for gaming and even programming itself.

If you enjoyed what you read, click here to try *Coding Through the Ages* for yourself.

Appendix A: Survey Questions

The following pages are the survey that we had playtesters fill out at the end of their session. They contain both multiple choice and free response questions about background and game experience, specifically regarding the visuals. The UUID at the end of the survey was randomly generated and and not editable by the user. This was used to link the survey responses to the player's last run of the game, which had the same UUID.

Real-Time Feedback in Coding Based Games

In this survey, you will be prompted with several questions regarding your background skills and your gameplay experience. The survey should take only a few minutes to complete. Click Next to start the survey.

Skip to question 1Skip to question 1

Background Experience

Please answer the following questions honestly and to the best of your ability.

1. What is your programming skill level?

Mark only one oval.

- ◯ None
- ◯ Beginner (very basic programs, not too much coding experience)
- ◯ Intermediate (more complex programs, good amount of coding experience)
- ◯ Expert (advanced programs / software, lots of coding experience)

2. How fast of a reader are you?

Mark only one oval.

- ◯ I am a very slow reader, I need to take my time.
- ◯ I read at a reasonable pace and can get the gist of text fairly quickly.
- ◯ I am a fast reader, I can skim text and understand it clearly.

3. How well would you say you perform under pressure?

Mark only one oval.

- ◯ I do not work well under pressure, I get too stressed out.
- ◯ I can perform fairly well under pressure, with some very minor setbacks.
- ◯ I can still complete tasks extremely well under pressure.
- ◯ I work better under pressure than without having some constraint.

4. Before playing the game, how familiar were you with visual feedback while programming?

Mark only one oval.

- ◯ I had no idea visual feedback existed in the context of programming.
- ◯ I only knew a little bit about visual feedback, such as color coding in coding environments.
- ◯ I've seen and/or utilized several instances of visual feedback while coding, similar to the ones found in this game.

Skip to question 5

Gameplay Experience

Please answer the following questions honestly and to the best of your ability.

5. The instructions / explanations / tutorial provided me with enough information to understand all aspects of the game.

Mark only one oval.

	1	2	3	4	5	
Strongly disagree	◯	◯	◯	◯	◯	Strongly agree

6. I had enough time to complete different tasks in the game, such as making decisions, checking the manual, and creating automated functions.

Mark only one oval.

	1	2	3	4	5	
Strongly disagree	⬭	⬭	⬭	⬭	⬭	Strongly agree

7. Did you feel comfortable coding in the environment provided in the game, or were there any restrictions that bothered you? Please explain.

8. I had a good understanding of how each resource changed over time.

Mark only one oval.

	1	2	3	4	5	
Strongly disagree	⬭	⬭	⬭	⬭	⬭	Strongly agree

9. It was clear to see the results of my decisions and how they affected the overall gameplay.

Mark only one oval.

	1	2	3	4	5	
Strongly disagree	⬭	⬭	⬭	⬭	⬭	Strongly agree

10. Did you feel that the results of your decisions were fair, and did they make sense in the context of the story? Please explain.

11. How often did you look at past decisions to influence your current choice?

Mark only one oval.

	1	2	3	4	5	
Rarely	◯	◯	◯	◯	◯	Always

12. Briefly, how would you describe your overall gameplay experience, especially in regards to the visuals?

13. Did you discover any bugs with the game? Please describe them in detail.

Skip to question 14

ADMIN ONLY

If you are not an admin, do not change this field. This question will be automatically entered as the player goes from the game to this survey. This will allow the player's game data to be linked to their responses here.

14. UUID

____

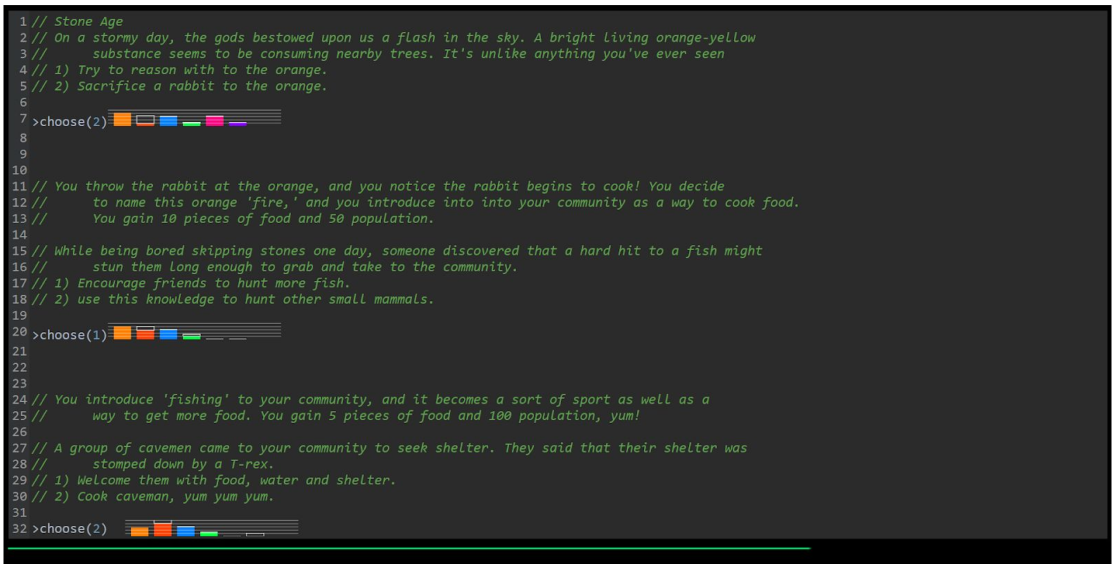
Appendix B: Study Protocols

The following pages are for the study protocol we made when submitting our playtesting proposal to the IRB. It outlined the premise of the game, our playtesting setup and procedure, and what data we would be collecting from players.

Realtime Feedback in Coding-Based Games
Standard Playtesting Protocols

Game Summary:

In this coding-based simulation game, the player must type code to make decisions to balance different resources such as hunger, population, and science that they are trying to manage. The game sees the player go through an entire timeline of human history, starting with the prehistoric era and ending in a futuristic age. Within each age, the player is presented with a challenge that they must solve by choosing one of two options, and each option will change the amount of resources they have in different ways. The player can code different commands to control the game, such as choose() to make a decision, eat() to eat a piece of food to keep their hunger up, and secure() to raise the security of the system they are coding with. There is also a special command called automate() that allows players to code functions that are run depending on the conditions within the function they specify. The player wins by making it until the end of the futuristic age, and the ultimate goal is to keep all of the resources balanced. If the player's hunger reaches zero before the end of the game, the game is over.



```
// Stone Age
// On a stormy day, the gods bestowed upon us a flash in the sky. A bright living orange-yellow
//      substance seems to be consuming nearby trees. It's unlike anything you've ever seen
// 1) Try to reason with to the orange.
// 2) Sacrifice a rabbit to the orange.

>choose(2)



// You throw the rabbit at the orange, and you notice the rabbit begins to cook! You decide
//      to name this orange 'fire,' and you introduce into into your community as a way to cook food.
//      You gain 10 pieces of food and 50 population.

// While being bored skipping stones one day, someone discovered that a hard hit to a fish might
//      stun them long enough to grab and take to the community.
// 1) Encourage friends to hunt more fish.
// 2) use this knowledge to hunt other small mammals.

>choose(1)



// You introduce 'fishing' to your community, and it becomes a sort of sport as well as a
//      way to get more food. You gain 5 pieces of food and 100 population, yum!

// A group of cavemen came to your community to seek shelter. They said that their shelter was
//      stomped down by a T-rex.
// 1) Welcome them with food, water and shelter.
// 2) Cook caveman, yum yum yum.

>choose(2)
```

Setup:

For the experiment, we will host several sessions over Zoom to allow us to observe the gameplay via screen sharing as the participant plays it and to hear the player's immediate responses to the game elements. Besides having a stable Internet connection, the only other setup consists of electronic copies of the informed consent form to send to the participant, the link to the website where the game is hosted, and a Google Form survey that the player will fill out at the end of the session.

Procedure:

The general procedure of the playtesting session will be as follows:

- Players will sign up for playtesting with their emails in a spreadsheet that the hosts will send out a week before playtesting begins.
- The hosts will send out an invitation link for the Zoom session after sign ups are completed. The player will join the Zoom call at the time they signed up for.
- At least two group members, one to lead the session and one to take notes, will host the Zoom session. When the player arrives, the hosts will send them an electronic copy of the informed consent form. After reading through the document, the player can decide to leave without signing and participating (and NOT receive the $10 eGift card), or they can stay and provide an electronic signature at the bottom of the consent form.
- If they decide to stay, the hosts will send them the URL to the online website where the game is hosted, and the player can begin playing. Anonymous data will be collected during gameplay (see the Data Collection section below for more details).
- The player will be able to go through the game as many times as they like, even if they get a game over at any point or complete the game early. The session will be a total of 30 minutes, and the players have up until 5 minutes at the end of the session to keep playing, at which point the hosts will redirect the player to the survey. The player can also decide to skip directly to the survey at any point when the game ends (game over or winning).
- When they finish their game session, the player will complete the anonymous survey, which consists of questions about the game mechanics and their experience playing the game.
- Before leaving the playtesting session, the host will describe the next steps in terms of receiving compensation. Later that day after the session, an email will be sent to the player thanking them for testing the game. By early May, they will receive another email with a $10 Amazon eGift card. The player will then leave the session.

Data Collection:

We will collect the following data from each player during the course of the game:

- The time for each player to make each decision
- The values of each statistic made at each decision and at the end of each age
- Think-aloud notes during the game session
- An anonymous, unique user ID generated by the program

We will also be asking questions in the surveys about topics such as coding ability, opinions on inline visuals in coding, and if the game was enjoyable. We will NOT be collecting any sensitive data that can be directly linked back to any individual, except for the player's email. The email will be used for sending out Zoom invitation links after signing up, when we send an email thanking the player for participating after the sessions that day, and when sending an email by early May with their $10 Amazon eGift card. However, their name and email will NOT be linked to their gameplay data and will not be collected by the program.

We will be collecting this data within a script that runs the game and sending them to a Firebase database that we will then look through later to conduct some data analysis on and create some charts and graphs. No personal identifiable information, such as name and email, will be presented to the public with any of this data, so they will remain anonymous.

APPENDIX C: INFORMED CONSENT FORM

The following pages are the informed consent form that we required players to read and sign before continuing with each playtesting session. This was required by the IRB, as even projects involving human subjects with minimal risk must have a consent form. It details the purpose of playtesting, the testing procedures, the risks and compensation involved, and a place at the bottom for both the participant and host to sign.

Informed Consent Agreement for Participation in Playtesting

Investigators: John Amaral, Shuxing Li, Grace Pelella

Contact Information: gr-rfinccmqp@wpi.edu

Title of Study: Real-time Feedback in Coding-Based Games

Advisors: Lane Harrison and Charlie Roberts

Introduction: You are being asked to participate in a game playtesting session. Before you agree, however, you must be fully informed about the purpose of the study, the procedures to be followed, and any benefits, risks or discomfort that you may experience as a result of your participation. This form presents information about the study so that you may make a fully informed decision regarding your participation.

Purpose of the study: The study aims to understand if players believe whether the presence of graphical visuals allows them to fulfill the goals of the game or if they are more distracting, as well as to observe whether the game provides an interesting and enjoyable experience or not.

Procedures to be followed: The following steps will be conducted sequentially to ensure a valuable experience for both the players and the host:

At the scheduled time, the player will join the Zoom link that was sent out via email by the hosts. Once the player has joined the Zoom call, the hosts will share an electronic copy of this consent form, and the player will read this form in its entirety. **Upon reading the form and asking the hosts any questions or bringing up any concerns, the player can choose to no longer participate and leave the Zoom call. However, the Amazon $10 eGift card will NOT be awarded to players who do not sign this consent form and play the game at least once through (either getting a game over or winning the game).**

If the player does decide to stay, they need to sign this consent form at the bottom, and the hosts must sign as well. After signing, the hosts will send the player a GitHub pages link to the game, which will be played in a web browser of choice. (For the best results, Google Chrome is recommended, although any other browser is acceptable. A stable Internet connection is also necessary for data collection.) The game will then load in the web browser, and the player will share their screen with the hosts so the hosts can observe their gameplay. They will also be encouraged to think aloud and describe any criticisms or observations to the hosts as they experience the game. The player can ask questions during the game, although the answers from the hosts may be limited to let the players discover gameplay for themselves.

During the course of the game, the player's gameplay data will be collected, but it will remain completely anonymous and in no way will have any personal identifiable information. The player will have most of the remaining 30-minute session to play through the game. The player can restart the game as many times as they like, and at any point they can decide to stop playing and fill out the survey. Otherwise, once 5 minutes of the session is left, the hosts will direct the player to the survey to have enough time to complete it.

The survey will also be anonymous, but it will have a special ID that can link it to the previously-collected gameplay data. The player can (and it is recommended that they do) stop sharing their screen upon completion of the survey to keep the responses anonymous. Once the player submits the survey, the hosts will inform them of two emails being sent to them: one later that day thanking them for their playtesting participation, and another email with their awarded $10 Amazon eGift card that will be sent out by early May. After this brief summary, the player can leave the session.

Risks to players: Although there are no major risks to players during this game, **they will need a good Internet connection to be able to participate**. Having a stable Internet connection will ensure that the player can share their screen over Zoom and play through the game at least once while their data is being collected so they can be awarded the $10 Amazon eGift card via email by early May.

Benefits to players and others: The player will receive a $10 Amazon eGift card in an email sent out by early May, and they may also gain some useful programming skills by participating in the playtesting session. <u>The $10 Amazon eGift card will only be emailed if the player completes the experiment by playing through the game at least once (whether winning the game or receiving a game over).</u>

Record keeping and confidentiality: All data recorded will be anonymous. Only the game developers will have access to the exact data collected during and after the platesting session. The data will be stored in a cloud database that is only accessible to the developer team. Records of player participation in this study will be held confidential so far as permitted by law. The study investigators, the sponsors under certain circumstances, the Worcester Polytechnic Institute Institutional Review Board (WPI IRB) will be able to inspect and have access to player data that does not identify the player by name.

However, we will be collecting the name and email of each player during the playtesting session, only for the purposes of playtesting session signups, sending a confirmation email after the session thanking the player for their time, and sending another email by early May with the $10 Amazon eGift card. The IMGD department in charge of funding may also need to know the emails to have the eGift cards properly distributed. However, **any publication or presentation of the data will NOT identify the player.**

Compensation or treatment in the event of injury: The playtesting session does not have any physical risk to the participants. The player does not give up any of their legal rights by signing this statement. If an unpredicted impact in Internet connection occurs during the session, if the player already played through the game at least once, they will still receive the $10 Amazon eGift card via email by early May. However, if an interruption early on in the session prevents any data from being collected and prevents the player from completing the game even once, the hosts will NOT compensate the participant with the gift card. This makes the necessity for a stable Internet connection all the more important, and that should be carefully considered before signing below.

Cost/Payment: The playtesting session will not cost any money to participate in, but there will be a $10 Amazon eGift card sent in an email to all participants by early May (upon completion of the game once, whether by receiving a game over or winning the game).

For more information about this research or about the rights of research participants, or in case of research-related injury, contact:

Developer contact: gr-rfinccmqp@wpi.edu
IRB Manager: Ruth McKeogh, Tel. 508 831- 6699, Email: irb@wpi.edu
Human Protection Administrator: Gabriel Johnson, Tel. 508-831-4989, Email: gjohnson@wpi.edu

Your participation in this playtesting session is voluntary. Your refusal to participate does mean that you will not receive the $10 Amazon eGift card, but there is no other penalty to deciding not to participate. You may also decide to stop participating in the session at any time, but as previously stated, you will NOT receive the $10 eGift card unless you complete the game once, whether ending by game over or by winning. The project investigators retain the right to cancel or postpone the session at any time they see fit. However, if it is done during the session and the player has gone through the game at least once (and some of their data has been collected), the player will still receive the $10 eGift card.

By signing below, you acknowledge that you have been informed about and consent to be participating in the playtesting session described above. Make sure that your questions are answered to your satisfaction before signing. You are entitled to retain a copy of this consent agreement.

______________________ Date: ________________

Study Participant Signature

____

Study Participant Name (Please print)

____________________________________ Date: ________________

Signature of Person who explained this study

Appendix D: Crowdsourcing

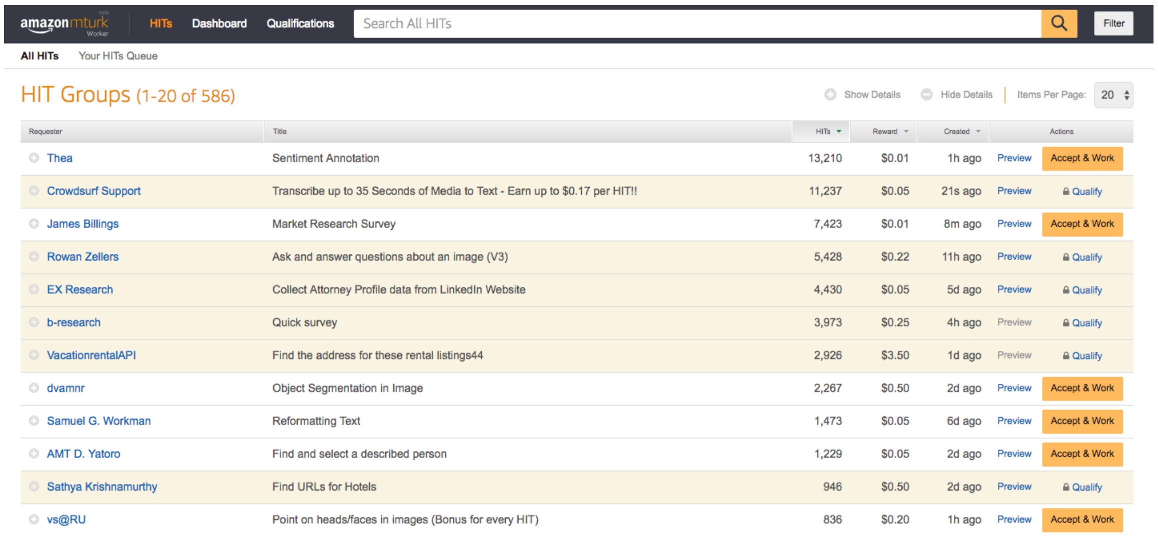
The following writing was originally part of our Background section, as earlier we intended to utilize crowdsourcing in an online experiment setup. However, due to wanting to move towards playtesting rather than an official experiment, we took the research out of the Background and placed it here.

Experimentation is an absolutely necessary step in justifying a hypothesis, and acquiring meaningful results is the backbone of successful experimentation. Classic experiments utilized in-person studies to directly gain feedback from participants as the study was conducted, and many experiments today still use this method. It can be very reliable in terms of obtaining immediate user feedback by having the user “think aloud” as he or she is going through the experiment. However, this method is not cost or time effective: several expenses are needed to set up the experiment in-person, and enough time has to be spent on each participant completing a set number of tasks for the experiment.

The method of crowdsourcing aims to reduce these issues in experimentation. Crowdsourcing is an online method of receiving feedback from a wide range of users. Through crowdsourcing, the cost for experimentation is reduced significantly: instead of paying for physical equipment or hiring professionals, as they are no longer needed, the cost is mainly spent on paying the study participants for completing assigned tasks. Time is also not an issue, as most experiments are broken down into sub-tasks that can be completed in a matter of minutes, and infinitely many participants can complete the same task at once [13].

There are a few drawbacks to crowdsourcing, however. As Judith Aquino notes in her observations of crowdsourcing, there are two potential problems that can be introduced: lack of in-person communication and lack of credible results. Experiments that are best conducted with immediate feedback from participants may not suit the crowdsourcing formula too well, as human to human interaction can be quite limited online. Also, since crowdsourcing enables many more users to participate in a study, there is a larger focus on quantity over quality, meaning there could easily be some questionable responses / feedback mixed in with actual good results [16].

Depending on the type of study being conducted, the pros of crowdsourcing may outweigh the cons. For instance, Jeffrey Heer and Michael Bostock held an online graphical perception experiment using Amazon Mechanical Turk. MTurk allows results to be collected from a “global workforce” consisting of people called “Turkers” to complete each sub tasks in a study. For each sub task completed, a Turker is paid a certain amount of money determined by the organizers of the study, usually around 10 cents per task. **Figure 10** below shows a sample screenshot from Amazon Mechanical Turk [13].



HIT Groups (1-20 of 586)

Requester	Title	HITs	Reward	Created	Actions	
Thea	Sentiment Annotation	13,210	$0.01	1h ago	Preview	Accept & Work
Crowdsurf Support	Transcribe up to 35 Seconds of Media to Text - Earn up to $0.17 per HIT!!	11,237	$0.05	21s ago	Preview	Qualify
James Billings	Market Research Survey	7,423	$0.01	8m ago	Preview	Accept & Work
Rowan Zellers	Ask and answer questions about an image (V3)	5,428	$0.22	11h ago	Preview	Qualify
EX Research	Collect Attorney Profile data from LinkedIn Website	4,430	$0.05	5d ago	Preview	Qualify
b-research	Quick survey	3,973	$0.25	4h ago	Preview	Qualify
VacationrentalAPI	Find the address for these rental listings44	2,926	$3.50	1d ago	Preview	Qualify
dvamnr	Object Segmentation in Image	2,267	$0.50	2d ago	Preview	Accept & Work
Samuel G. Workman	Reformatting Text	1,473	$0.05	6d ago	Preview	Accept & Work
AMT D. Yatoro	Find and select a described person	1,229	$0.05	2d ago	Preview	Accept & Work
Sathya Krishnamurthy	Find URLs for Hotels	946	$0.50	2d ago	Preview	Qualify
vs@RU	Point on heads/faces in images (Bonus for every HIT)	836	$0.20	1h ago	Preview	Accept & Work

Figure 10: *An example screenshot from Amazon Mechanical Turk [13].*

Heer and Bostock found that crowdsourcing greatly reduced the cost of experimentation, promoted a faster completion rate of experiment tasks, and allowed for a more diverse pool of responses. Their experiments consisted of comparing differences in the visual spacing, contrast, and size of all elements of graphs. Heer and Bostock replicated a previous study that was originally completed in-person and compared their crowdsourcing results to the original study. **Figure 11** below shows an example of one of the experiments that Heer and Bostock conducted [13].

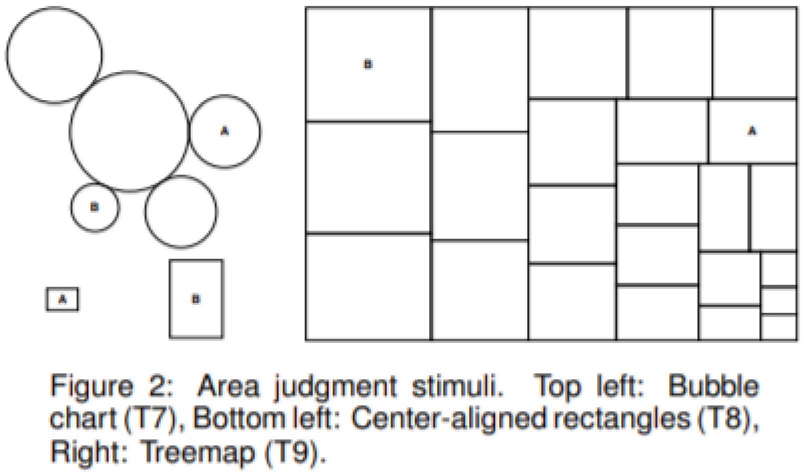


Figure 2: Area judgment stimuli. Top left: Bubble chart (T7), Bottom left: Center-aligned rectangles (T8), Right: Treemap (T9).

Figure 11: *An example of one of Heer and Bostock's experiments where different screen webpage layouts were tested [13].*

Surprisingly, most of their results were similar to those of the original study, and in some cases the variation in results was even less than the original study. Overall, they discovered that crowdsourcing was an extremely viable method of conducting studies, and in combination with traditional, in-person studies could lead to very important results [13].

There are many other crowdsourcing options to choose from besides MTurk, as crowdsourcing has become much more popular in recent years. Another popular crowdsourcing platform is Prolific [14]. Prolific is similar to MTurk in that there is a collective group of workers dedicated to completing studies online. However, unlike MTurk, Prolific has multiple payment options and does not pay per subtask of a study. Prolific allows either hourly payment, in which it suggest 6.50 per hour, or payment per study. Payment per study depends upon the number of participants required and the estimated time for each participant to complete the study, which can be calculated on the Prolific website. For instance, for 50 participants and 1 minute for each to complete the study, the estimated cost is 7.34 per study. Prolific is slightly more expensive than MTurk overall, but it claims to have even higher quality results due to verification and monitoring of participants. **Figure 12** shows a sample screenshot from Prolific's website.

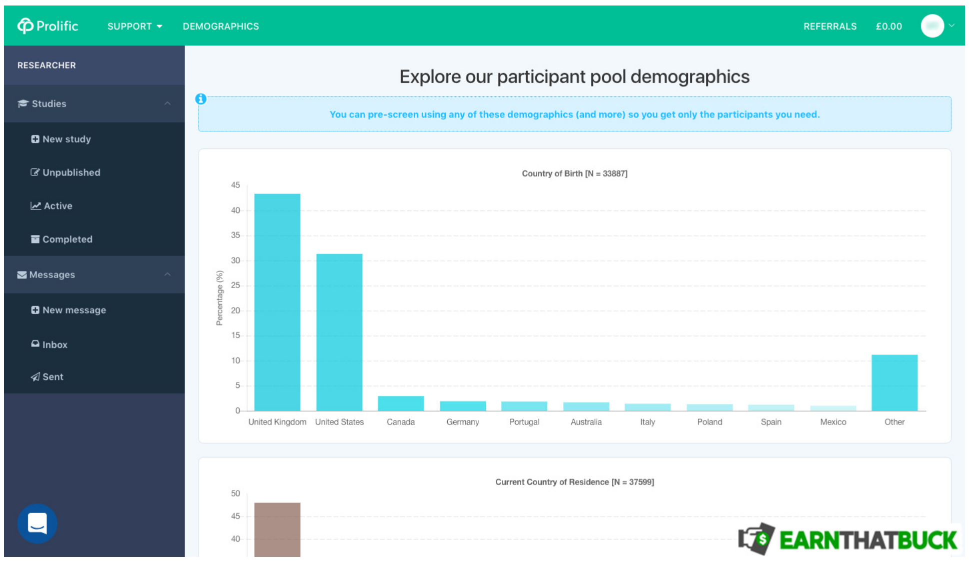


Figure 12: *A sample screenshot from Prolific [14].*

Since crowdsourcing has become an extremely promising method of gathering experiment results in recent years, the options for crowdsourcing has also increased. Almost everyone has access to the Internet, and when in-person equipment and people are not accessible, crowdsourcing provides an easy way to collect lots of results with a certain degree of quality. These results can be gathered at relatively low costs in a short amount of time. Although there may be lesser quality in results and a lack of person-to-person communication with crowdsourcing, on its own it is still an effective way of gathering data, and in combination with in-person studies can lead to impressive results.

Bibliography

[1] A. Sorensen, "Impromptu heads up display," jun 2011. [Online]. Available: https://vimeo.com/25699729

[2] F. Lantz, "Universal paperclips," Online Browser Game, 2017. [Online]. Available: https://www.decisionproblem.com/paperclips/index2.html

[3] B. Victor. (2012, sep) Learnable programming. [Online]. Available: http://worrydream.com/#!/LearnableProgramming

[4] C. Roberts. (2018, nov) Realtime annotations & visualizations in live coding performance. [Online]. Available: https://charlieroberts.github.io/annotationsAndVisualizations/

[5] D. Digital, "Reigns," Video Game, aug 2016.

[6] J. Hoffswell, A. Satyanarayan, and J. Heer, "Augmenting code with in situ visualizations to aid program understanding," University of Washington, Standford University, Tech. Rep., apr 2018.

[7] *Light Table*, Kodowa Inc., 2014.

[8] B. Swift, A. Sorensen, H. Gardner, and J. Hosking, "Visual code annotations for cyberphysical programming," in *2013 1st International Workshop on Live Programming (LIVE)*, 2013, pp. 27–30.

[9] Infocom, "Suspended," Computer Game, 1983.

[10] (2020) Codingame. [Online]. Available: https://www.codingame.com/start

[11] Apple. (2020) Swift playgrounds. [Online]. Available: https://www.apple.com/swift/playgrounds/

[12] Microsoft, "Kodu game lab," Computer Program, 2018.

[13] J. Heer and M. Bostock, "Crowdsourcing graphical perception: Using mechanical turk to assess visualization design," Stanford University, Tech. Rep., apr 2010.

[14] (2020) Prolific. [Online]. Available: https://www.prolific.co/

[15] M. Haverbeke, *CodeMirror*, apr 2020.

[16] J. Aquino, “The pros and cons of crowdsourcing,” *CRM Magazine*, vol. 17, p. 30, feb 2013.